SEMANTIC AND LOGICAL FOUNDATIONS FOR RANDOMIZED CONCURRENT PROGRAMS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Noam Zilberstein

August 2026

© 2026 Noam Zilberstein

ALL RIGHTS RESERVED

SEMANTIC AND LOGICAL FOUNDATIONS FOR RANDOMIZED CONCURRENT PROGRAMS

Noam Zilberstein, Ph.D.

Cornell University 2026

This thesis explores semantic foundations and logical techniques to reason about programs that are both randomized and concurrent. Concurrent programs display unpredictable behavior due to the different ways that the scheduler can interleave parallel threads. Randomness prevents the program's flow from being exploited by the scheduler to force undesirable outcomes. For decades, the combination of randomization and concurrency has been used to design consensus and synchronization protocols, ensure security and privacy of data, and generally improve the efficiency and reliability of distributed systems.

Due to the complex ways that concurrency and randomization interact, ensuring the correctness of programs that make use of both effects through testing and informal means is intractable, motivating the use of formal methods instead. However, prior to the work presented in this thesis, there was no semantic model that could capture the distributions of outcomes in randomized concurrent programs with unbounded loops, let alone logical techniques to reason about them.

This thesis introduces a program logic called Probabilistic Concurrent Outcome Logic (PCOL), which provides a powerful toolset for verifying the correctness of randomized concurrent programs. Although the construction of PCOL is quite sophisticated, this thesis works from first principles to thoroughly explain the metatheory of PCOL in a self contained way. Inspired by separation logic, PCOL uses compositional reasoning techniques to derive expressive specifications without considering all the possible interleavings of concurrent threads. That expressive power is demonstrated by proving correctness of a variety of case studies including randomness preserving protocols, a concurrent shuffling algorithm, and private information retrieval.

BIOGRAPHICAL SKETCH

I was born in Berkeley, California. Shortly thereafter, my family moved to Amherst, Massachusetts where I spent most of my childhood, apart from a year in Haifa, Israel. I graduated *summa cum laude* with a Bachelor of Science in Computer and Information Science from the University of Pennsylvania in 2015. After completing my undergraduate studies, I worked as a software engineer at Facebook in London, UK and Menlo Park, California for six years. I started graduate school at Cornell in September 2021.

This dissertation is dedicated to my grandparents.

Saba Arie (ז״ל), whose education was cut short by the Holocaust, nevertheless displayed an amazing engineering acumen. From him, I inherited the ability to figure out how things work.

Grandpa Sheldon, the first of many academics in the family, instilled intellectual values in all of us. From him, I inherited the intellectual curiosity that sustained my doctoral research.

Savta Esther is always sure to use fresh and clean ingredients in her cooking. When teaching me how to make schnitzel, she told me:

"I never use store-bought breadcrumbs" "**אף פעם לא קניתי פירורי לחם**"

That sentiment inspired me to always work from first principles, resulting in more elegant mathematical constructions. Nothing in this thesis is store-bought.

Grandma Barbara passed on to me her hunger for meat (that is, the important stuff), and also the tenacity to get it. That hunger kept me focused on impactful research throughout my studies.

ACKNOWLEDGEMENTS

A great number of mentors, colleagues, friends, and family members have helped me to arrive at this moment. My path towards earning a PhD was not a straight line; I took a significant detour into industry along the way. While I have the utmost passion for the content of this dissertation, it was never inevitable that I would devote my career to studying programming languages (PL).

Ironically, I was never particularly interested in programming. After an unpleasant experience with Java in high school, I applied to college as a Mechanical Engineering major, which was much more in line with my real passion: cars. On my first day of class I met Joe Schaffer, who shared my passions for mechanical engineering, but was also curious about computer science. Through Joe, I started to understand that computing is not just about programming in Java. I realized that I would be more fulfilled learning about algorithms, proof assistants, and monads rather than thermodynamics and vibrations, so I switched majors and never looked back.

In my later years at Penn, I was fortunate to take courses in advanced functional programming, automated theorem proving, and compilers with Stephanie Weirich and Steve Zdancewic. Those courses gave me a taste of cutting edge PL topics. For my *Senior Design Project*, I teamed up with my friends Lucas Peña, Tate Mandel, and Tiernan Garsys to prove that Haskell's type system is sound. We were advised by Stephanie Weirich and Richard Eisenberg, who patiently got us up to speed on the underlying mechanisms behind Haskell's latest and greatest features.

Upon graduating from Penn, I knew that I wanted to go to graduate school, but I needed a break from academia to try something new. I moved to London to work at Facebook, building infrastructure for spam fighting in Haskell. During my time at Facebook, I was eager to apply everything that Stephanie has taught me about dependently typed Haskell programming [Zilberstein 2020], egged on by Simon Marlow and my other teammates. Although I had initially planned to stay for only a year or two before applying to PhD programs, I was having so much fun that the prospect of getting a PhD faded.

But as I rose up through the ranks, my job became less about experimenting with dependently

typed programming and more about the actual priorities of my team: fighting spam and abuse. While that is important, it offered little intellectual stimulation in the way of deep PL problems. So, after four years, I started looking for a new challenge that would align more with my interests.

Nick Benton recruited me to the Programming Languages and Runtimes (PL&R) team to work on an experimental information flow type system, which would help to enforce privacy policies in Facebook's front end code. The project was a perfect fit. I also got to work with Mistral Contrastin and Quentin Carbonneaux, two PL PhDs who taught me about interesting new ideas every day. Despite rapid progress, a promising proof of concept, and a successful demo with our VP, the project ultimately never made it to production due to shifting priorities within the company.

In searching for something new to work on, I discovered an opportunity to apply for a *hack-a-month* that would be overseen by the CTO. I wanted to propose a verification project, so Nick put me in touch with Peter O'Hearn, who suggested working with a group of OS engineers keen on applying formal methods to their new microkernel. We never heard back from the CTO, but the project was so juicy that we just had to give it a shot. In exploring how to get started, I met Derek Dreyer, who gave us some encouragement that the project would be feasible with Iris. Once again working with Quentin, we succeeded in verifying the interprocess communication mechanism of the microkernel, and even found a few bugs in the process [Carbonneaux et al. 2022]. That project is still one of the largest industrial success stories using Iris.

Despite my formal methods escapade, there weren't opportunities at Facebook to do the kind of foundational research that I yearned for. I thought back to what my former manager Gemma Silvers once asked me: *"do you want to be right, or do you want to get things done?"* Of course, the answer that would advance my career was that I wanted to get things done. But engineering never fully satisfied me because it didn't give me the opportunities to dig deep into conceptual material and generate novel intellectual value. It was time for me to pivot and get back into academia. As Peter put it, *"Do you want to get this proof done, or be right? Wait, you can do both!"*

I applied to Cornell not knowing that Alexandra Silva would be moving there that very same

year. In that sense, it feels like fate she ended up as my advisor; sometimes things just work out. Alexandra mostly just let me do my own thing, which is exactly what I wanted after six years of aligning my work with *business value*. But I can't thank her enough for the endless support and opportunities that she provided for me. With a small push, she sent me down an endless path of discovery where new insights about correctness, incorrectness, nondeterminism, and randomization led me to develop the generalized theory that became Outcome Logic.

Along the way, I met many amazing collaborators. Derek Dreyer was eager to explore alternative approaches to incorrectness and taught me all about academic writing. Joe Tassarotti helped me get up to speed on randomized concurrency and gave me an academic home in New York City. Daniele Gorla sacrificed his winter break to frantically prove properties about infinite pomsets with me. My committee members Dexter Kozen, Adrian Sampson, and Robbert van Renesse pointed me towards concurrency during my qualifying exam, which turned into this dissertation.

I come from a very academic family and my parents were always a bit skeptical that I would survive a full career in industry. But rather than meddling with my trajectory, they let me come to the conclusion that I should do a PhD on my own. I appreciate their support in all of my endeavors. My wife Sarah got her PhD in English Literature in 2024, my sister Shira finished her PhD in Sociology about a month before I did, and my brother Itai started his PhD the previous fall. Surrounded by academics, I have benefited from an intellectually stimulating environment, and also from the support of people who understand the challenges of graduate school.

I have to especially thank Sarah, who endured the brunt of my *kvetching* after every paper rejection and throughout the ups and downs of my job market experience. Sarah always had confidence that I would succeed, but also frequently reminded me that there is more to life than papers and proofs. Last, but not least, I have thank our dog Penny. At the times when I felt the most stuck in the weeds with some proof, it often turned out that taking her out for a walk in the fresh air was exactly the catalyst that I needed in order to make a breakthrough. It just goes to show that sometimes inspiration comes from the most unexpected places.

TABLE OF CONTENTS

Biographical Sketch . . . iii
Dedication . . . iv
Acknowledgements . . . v
Table of Contents . . . viii
List of Figures . . . ix

1 Introduction **1**
1.1 Theoretical Challenges . . . 2
1.2 Key Ideas and Contributions . . . 4
1.3 Outline and Previously Published Work . . . 10

2 Preliminaries and Mathematical Foundations **16**
2.1 Denotational Semantics and Domain Theory . . . 16
2.2 Probabilistic Programming . . . 27
2.3 Hoare Logic . . . 33
2.4 Separation Logic . . . 35
2.5 Outcome Logic . . . 40

3 A Measure Theoretic Model of Outcomes **43**
3.1 Overview . . . 43
3.2 Preliminaries . . . 48
3.3 Outcome Assertions . . . 54
3.4 Related Work . . . 59

4 Probabilistic Outcome Logic **61**
4.1 Overview . . . 61
4.2 Formalizing Outcome Triples . . . 65
4.3 Proof Rules . . . 66
4.4 Examples . . . 71
4.5 Related Work . . . 74

5 Demonic Outcome Logic **79**
5.1 Overview . . . 79
5.2 Denotational Semantics and the Convex Powerdomain . . . 85
5.3 Demonic Outcome Assertions . . . 92
5.4 Demonic Outcome Logic . . . 95
5.5 Examples . . . 101
5.6 Related Work . . . 105

6 Denotational Semantics for Probabilistic Concurrent Programs **111**
6.1 Overview . . . 111
6.2 Semantic Structures . . . 115
6.3 Pomset Semantics . . . 121
6.4 Linearization and Adequacy . . . 127
6.5 Related Work . . . 131

7 Probabilistic Concurrent Outcome Logic **133**
7.1 Overview . . . 133
7.2 Invariant Sensitive Semantics . . . 139
7.3 Probabilistic Concurrent Outcome Logic . . . 143
7.4 Soundness . . . 152
7.5 Examples . . . 156
7.6 Related Work . . . 160

8 Case Studies **163**
8.1 The Entropy Mixer . . . 163
8.2 Concurrent Shuffle . . . 168
8.3 Private Information Retrieval . . . 175

9 Conclusion **181**
9.1 Further Theoretical Developments . . . 181
9.2 Mechanization and Automation . . . 186
9.3 Applications . . . 188

Bibliography **190**

LIST OF FIGURES

1.1 A concurrent shuffling algorithm. 5

2.1 Syntax of Command Language (CL) programs, where $a \in act$. 17
2.2 Denotational semantics of basic CL programs. 27
2.3 Syntax of Probabilistic Command Language (pCL) programs. 28
2.4 Denotational semantics $[\![-]\!]_{\mathcal{D}} : cmd_{\mathrm{P}} \to \mathsf{Mem}[V] \to \mathcal{D}(\mathsf{Mem}[V]_{\perp})$ of the Probabilistic Command Language (pCL). The semantics of all commands except for the new probabilistic ones $C_1 \oplus_e C_2$ and $x :\approx d(e)$ carry over from Figure 2.2. . 29
2.5 The inference rules of Hoare Logic, where $P[e/x]$ denotes syntactic substitution of the expression e for the variable x in the assertion P. 34
2.6 Satisfaction relation for separation logic assertions. 37

3.1 Outcome Assertion syntax. 54
3.2 The satisfaction relation for outcome assertions, where $\Gamma : \mathsf{LVar} \to \mathsf{Var}$ is a logical context and $\mathcal{P} = \langle \mathsf{Mem}[V], \mathcal{F}_{\mathcal{P}}, \mu_{\mathcal{P}} \rangle$ is a complete probability space. All the existentially quantified probability spaces are also complete. 55
3.3 Selected entailment laws, where $\varphi[E/X]$ is a syntactic substitution of E for X in φ. 58

4.1 Rules for sequential commands in pOL. 67
4.2 Structural pOL rules. 68
4.3 A loop rule for almost-sure termination in pOL. 71
4.4 Almost sure termination proof for a simple random walk in pOL. 72
4.5 Correctness proof for the von Neumann trick in pOL. 74

5.1 Syntax of the Probabilistic Demonic Command Language (pdCL). The new additions over pCL are highlighted in yellow. As before, tests $b \in test$ and expressions $e \in exp$ are discussed in Section 2.1.1 and distributions d are from Figure 2.3 . . . 86
5.2 Denotational semantics $[\![-]\!]_{C} : cmd_{\mathrm{PD}} \to \mathsf{Mem}[V] \to C(\mathsf{Mem}[V])$ of pdCL. The semantics of standard commands carry over from Figure 2.2. 90
5.3 Outcome Assertion syntax extended with demonic constructs, where $m \in \{\mathrm{s}, \mathrm{w}\}$. . 92
5.4 Selected entailment laws with new assertions for nondeterministic reasoning. . . 94
5.5 Inference rules that carry over to dOL from pOL. 97
5.6 Framing rules for dOL. 98
5.7 dOL rules for reasoning about nondeterminism. 99
5.8 The loop analysis rule of dOL. 101
5.9 Almost sure termination proof for the adversarial random walk 103
5.10 Correctness proof for the adversarial von Neumann trick. 106

6.1 Syntax of the Probabilistic Concurrent Command Language (pcCL). The new addition over pdCL is highlighted in yellow. As before, tests $b \in test$ and expressions $e \in exp$ are discussed in Section 2.1.1, distributions d are from Figure 2.3, and actions are the same from Figure 5.1. 112

6.2 A randomized concurrent program (left) and its associated semantic structure (right). . . . 113
6.3 Pomset semantics of commands $[\![-]\!]_{pom} : cmd_{\text{PC}} \to pom$. . . . 122
6.4 Linearization for finite LPOFS $\mathcal{L}_{lpo}$: $lpo_{\text{fin}} \times form \times 2^{nodes} \to \mathsf{Mem}[V] \to C(\mathsf{Mem}[V])$. . . . 128

7.1 Sketch of the derivation for program (7.1). . . . 136
7.2 Inference rules for commands that carry over to PCOL from DOL. . . . 145
7.3 Structural rules that carry over to PCOL from DOL. . . . 146
7.4 Concurrent Separation Logic Rules 147
7.5 Almost sure termination proof for the concurrent random walk 157
7.6 Correctness proof for the concurrent von Neumann trick. . . . 159

8.1 Derivation for the first thread of the entropy mixer. . . . 164
8.2 Entropy mixer main derivation. . . . 166
8.3 A concurrent shuffling algorithm. . . . 168
8.4 Correctness proof for the $\mathbf{Shuffle}_k$ routine. . . . 170
8.5 Derivation for the loop body of **Partition**. . . . 172
8.6 Main derivation for the concurrent shuffle. . . . 174
8.7 A concurrent private information retrieval protocol. . . . 176
8.8 Correctness derivation for the $\mathbf{Fetch}_k$ procedure. . . . 177
8.9 Correctness derivation for the **PrivFetch** program. . . . 179

CHAPTER 1

INTRODUCTION

Randomization has long played a central role in concurrent computing, offering elegant and practical solutions to problems that are difficult, or impossible, to solve deterministically. The power of randomization was highlighted throughout the 1970s and 1980s when it was proven that some consensus and synchronization problems have no deterministic solution [Dijkstra 1971; Angluin 1980; Fischer et al. 1985], yet they have simple randomized ones [Lehmann and Rabin 1981; Ben-Or 1983; Itai and Rodeh 1990].

Randomization, via explicit coin flip operations performed in the program, breaks the symmetries that underlie those impossibility results and has thus become an essential tool in the design of concurrent algorithms. Beyond distributed protocols, randomization is also crucial to security and privacy and can improve algorithmic efficiency, even for problems that do have deterministic solutions. Concrete applications include distributed cryptography [Alpos and Cachin 2023], consensus [Gilad et al. 2017; Miller et al. 2016; Frank et al. 2025], databases [Corbett et al. 2012; Heule et al. 2013], and many others.

Despite their ubiquity, randomized concurrent algorithms are notoriously difficult to understand and test. Their behavior depends on both probabilistic choices and nondeterministic thread scheduling, leading to an enormous space of possible executions that cannot be exhaustively explored through testing or informal reasoning. Correctness proofs are a promising alternative, but the pen-and-paper approach results in proofs that are brittle and error prone, as Lehmann and Rabin remarked when studying the randomized Dining Philosophers problem:

> "*Proofs of correctness for probabilistic distributed systems are extremely slippery; in fact the proof presented here (hopefully correct) is only the last one in a sequence of incorrect proofs.*"
>
> — Lehmann and Rabin [1981]

Program logics offer a solution: by mechanically deriving correctness guarantees, they promise

not to further extend the "*sequence of incorrect proofs*." But in the forty-five years leading up to the publication of this thesis, formal logical techniques that augment concurrency reasoning with the ability to specify random outcomes have remained notably underdeveloped. Recent decades have seen major advances in logical techniques for concurrent programs and, separately, probabilistic programs, but a significant leap beyond the state of the art was still required in order to seamlessly integrate both paradigms into a single logical system.

This thesis introduces Probabilistic Concurrent Outcome Logic (PCOL), a foundational new approach to reasoning about programs that are both randomized and concurrent. The theory of PCOL is designed from the ground up to natively support both compositional concurrency reasoning and also expressive abilities to describe the distributions of random program behaviors. Such capabilities required fundamental advancements including a new semantic model called *pomsets with formulae*, and logical techniques to inspect the probabilistic outcomes from coin flip operations and the nondeterministic behaviors resulting from concurrent scheduling.

This thesis iteratively builds up the theory of PCOL from first principles, starting from purely probabilistic computation (Chapters 3 and 4), to adding explicit nondeterministic choice (Chapter 5), and finally to adding nondeterminism induced by concurrent scheduling (Chapters 6 and 7). Each progressive step introduces new challenges and insights, which are overviewed in the proceeding sections.

1.1 THEORETICAL CHALLENGES

Randomized concurrent algorithms feature two interacting types of unpredictable behavior. Random behaviors arise from explicit use of sampling operations $x :\approx d$, and can be analyzed quantitatively. Nondeterministic behaviors arise from concurrent scheduling $C_1 \parallel C_2$, and must be treated *demonically*, *i.e.,* the algorithm must operate properly even in the presence of an adversarial scheduler that always behaves in the worst way possible.

The interaction between probabilistic actions and nondeterministic scheduling results in com-

plex behavior. Consider the following program, which assigns a uniformly sampled Boolean to the variable x while the two concurrent threads race to assign y.

$$x :\approx \mathbf{Ber}\left(\tfrac{1}{2}\right) \fatsemi y := 1 \quad \| \quad y := 0$$

After executing the program, we may want to know the probability of x being equal to y. During execution, the scheduler will choose how to interleave the two threads, so the commands will execute in one of the following concrete orders.

$$x :\approx \mathbf{Ber}\left(\tfrac{1}{2}\right) \fatsemi y := 1 \fatsemi y := 0 \qquad x :\approx \mathbf{Ber}\left(\tfrac{1}{2}\right) \fatsemi y := 0 \fatsemi y := 1 \qquad y := 0 \fatsemi x :\approx \mathbf{Ber}\left(\tfrac{1}{2}\right) \fatsemi y := 1$$

However, we do not yet know the probability that each of those schedules could be chosen. Determining that probability requires us to carefully consider the capabilities of the scheduler. Can the scheduler *adapt* to probabilistically chosen values, or is it *oblivious* to the sampling operations [Ben-David et al. 1990]? Is the scheduler deterministic, or can it perform probabilistic computation too [Varacca and Winskel 2006]?

Once we answer those operational questions, we can develop a formal semantic model that is faithful to way that we expect programs to execute. Semantics gives us a ground truth meaning for each program, but it is too complicated to reason about directly. Logical rules on top of the semantics provide higher level ways to mechanically derive human-readable specifications about a program. A soundness proof connects the two by ensuring that each specification derived in the logic is valid according to the semantic model.

However, semantic models that combine randomization with a nondeterministic concurrent scheduler are extraordinarily complex. Several negative results have shown that naïve approaches to modeling the combination of nondeterministic and probabilistic computation are not well behaved [Varacca and Winskel 2006; Zwart and Marsden 2019; Zwart 2020; Parlant 2020], underscoring the need for a more rigorous approach. As a result, the *convex powerdomain* has emerged as a prominent formal model [He et al. 1997; McIver and Morgan 2005], but it had not previously been used to model concurrency.

Unbounded looping introduces further complexities. Many randomized algorithms do not terminate in a fixed amount of time. Instead, the probability of termination converges to one as time goes to infinity. Liveness analysis therefore requires complex probabilistic reasoning on infinite traces. Furthermore, bugs can be introduced when a correct algorithm is translated into a concrete implementation [Aguilera and Toueg 2012]. As a Google engineering team noted:

> "*In order to build a real-world system, an expert needs to use numerous ideas scattered in the literature and make several relatively small protocol extensions. The cumulative effort will be substantial and the final system will be based on an unproven protocol.*"
>
> — Chandra et al. [2007]

The more abstract the theory is, the harder it is to gauge whether an implementation matches it. Early attempts to prove probabilistic liveness modeled programs using Markov Decision Processes [Hart et al. 1983], which do not closely mirror the syntax of programs. Formal methods enable direct reasoning about programs that are syntactically similar to a concrete implementation, which helps prevent bugs that arise due to the mismatch between a program and its proof.

Building on decades of progress on formal methods for analyzing probabilistic programs and concurrent programs, this thesis presents the first semantic model (Chapter 6) and associated program logic (Chapters 3 to 5 and 7) for verifying randomized concurrent programs in terms of their distributions of outcomes. Prior approaches were not able to handle unbounded loops and has less expressive power to describe the distributions over observable behaviors, therefore examples like the one in Section 1.2 were previously out of reach of formal methods.

1.2 KEY IDEAS AND CONTRIBUTIONS

In order to demonstrate the novel features of pcOL, we now turn to the concurrent shuffling algorithm **`ConcurrentShuffle`** in Figure 1.1, which is a parallelized version of the Rao-Sandelius

$$\textbf{ConcurrentShuffle}:\ \textbf{Partition} \fatsemi (\textbf{Shuffle}_1 \parallel \textbf{Shuffle}_2) \fatsemi a \coloneqq a_1 +\!\!+ a_2$$

$\textbf{Partition}$:
$a_1 \coloneqq [] \fatsemi a_2 \coloneqq [] \fatsemi i \coloneqq 0 \fatsemi$
$\textbf{while}\ i < \textbf{len}(a)\ \textbf{do}$
$(a_1 \coloneqq \textbf{snoc}(a_1, a[i])) \oplus_{\frac{1}{2}} (a_2 \coloneqq \textbf{snoc}(a_2, a[i])) \fatsemi$
$i \coloneqq i + 1$

$\textbf{Shuffle}_k$:
$i_k \coloneqq \textbf{len}(a_k) \fatsemi$
$\textbf{while}\ i_k > 0\ \textbf{do}$
$j_k :\approx \textbf{unif}\ ([0, .., i_k]) \fatsemi$
$a_k \coloneqq \textbf{swap}(a_k, i_k, j_k) \fatsemi$
$i_k \coloneqq i_k - 1$

Figure 1.1: A concurrent shuffling algorithm.

[Rao 1961; Sandelius 1962] algorithm. Given an input list ℓ stored in the variable a, the goal of **ConcurrentShuffle** is to terminate in a state where a is a uniformly chosen permutation of ℓ.

The input list is first randomly bucketed into two sublists a_1 and a_2 using the **Partition** routine. Next, a_1 and a_2 are shuffled in parallel using the $\textbf{Shuffle}_k$ routine, which is an implementation of the Fisher-Yates algorithm [Fisher and Yates 1938]. Finally, the two segments are concatenated together to form the final shuffled list. A full explanation of the algorithm and accompanying correctness proof are given in Section 8.2, but we will now highlight some of the challenges that emerge in that proof to motivate the overall technical approach of this thesis.

1.2.1 Compositional Reasoning about Probabilistic Concurrency

When executing $\textbf{Shuffle}_1 \parallel \textbf{Shuffle}_2$, the scheduler chooses how to interleave the atomic commands in the two threads, both of which contain loops over lists of unspecified lengths. The concurrent execution therefore yields a massive number of possible interleavings, which grows exponentially with the size of the input list. It is not feasible to enumerate all of those executions, so we instead need a *compositional* way to reason about the two threads in isolation, and then make an inference about their behavior when composed in parallel.

Concurrent Separation Logic (CSL) [O'Hearn 2004; Brookes 2004] already provides compositional ways to reason about concurrent threads in non-probabilistic programs. Specifications in CSL are given in terms of preconditions and postconditions: $\{P\}\ C\ \{Q\}$ means that if P ini-

tially holds, then Q will hold after the command C runs. The Par rule below allows us to infer a specification about a concurrent program from specifications about the two individual threads.

$$\frac{\{P_1\}\ C_1\ \{Q_1\} \qquad \{P_2\}\ C_2\ \{Q_2\}}{\{P_1 * P_2\}\ C_1 \parallel C_2\ \{Q_1 * Q_2\}}\textsc{Par}$$

Above, the *separating conjunction* $P_1 * P_2$ means that P_1 and P_2 hold in disjoint portions of memory. That separation is crucial to the soundness of Par because it guarantees that the threads C_1 and C_2 cannot interfere with each other.

In pcOL, specifications are similarly given as triples $\langle \varphi \rangle\ C\ \langle \psi \rangle$, but the pre- and postcondition are now expressed in a probabilistic language. The assertion $\lceil P \rceil$ means that P holds with probability 1 and the *outcome conjunction* $\bigoplus_{X \sim d} \varphi$ means that the logical variable X is distributed according to the distribution d, with φ holding for every value of X. Combining the two, $x \sim d$ is syntactic sugar for $\bigoplus_{X \sim d} \lceil x \mapsto X \rceil$, which means that the value of x is distributed according to d. In the shuffle example, our goal is to derive the following specification, which states that if a initially holds the list ℓ, then a will be a uniform permutation of ℓ once the program terminates.

$$\langle \lceil a \mapsto \ell \rceil \rangle\ \texttt{ConcurrentShuffle}\ \langle a \sim \mathbf{unif}\ (\Pi(\ell)) \rangle$$

Let us now turn back to the parallel invocations of $\texttt{Shuffle}_1$ and $\texttt{Shuffle}_2$. As we previously established, the concurrent execution results in a massive blowup in execution paths, which cannot be exhaustively explored. We instead need a compositional approach where we reason about the threads in isolation and then infer their joint behavior. We start with specifications stating that each thread produces a uniform permutation of its input.

$$\langle \lceil a_1 \mapsto A_1 \rceil \rangle\ \texttt{Shuffle}_1\ \langle a_1 \sim \mathbf{unif}\ (\Pi(A_1)) \rangle$$

$$\langle \lceil a_2 \mapsto A_2 \rceil \rangle\ \texttt{Shuffle}_2\ \langle a_2 \sim \mathbf{unif}\ (\Pi(A_2)) \rangle$$

In addition, as we will see in Chapter 7, pcOL has a Par rule that allows us to combine the above specifications in a similar way to CSL, yielding the specification below.

$$\langle \lceil a_1 \mapsto A_1 \rceil * \lceil a_2 \mapsto A_2 \rceil \rangle\ \texttt{Shuffle}_1 \parallel \texttt{Shuffle}_2\ \langle a_1 \sim \mathbf{unif}\ (\Pi(A_1)) * a_2 \sim \mathbf{unif}\ (\Pi(A_2))) \rangle$$

But to complete the proof, it is not enough to know how a_1 and a_2 are distributed on their own; we also need to know their *joint distribution*. More precisely, if the scheduler could induce a *probabilistic correlation* such that observing a particular shuffle of a_1 makes some other shuffle of a_2 more likely, then the combinatorial argument would not work out to conclude that a is ultimately uniformly shuffled. We need to ensure that the concurrent execution results in a_1 and a_2 being *probabilistically independent*, which formally means that for any permutations π_1 and π_2 of the initial values of a_1 and a_2, the following equation holds at the end of the execution.

$$\mathbb{P}[a_1 = \pi_1 \land a_2 = \pi_2] = \mathbb{P}[a_1 = \pi_1] \cdot \mathbb{P}[a_2 = \pi_2]$$

The ability to recover the joint probability by multiplying the individual probabilities gives us the compositionality that we need to derive expressive specifications about parallel threads. Probabilistic independence is therefore one of the key concepts that makes concurrency reasoning tractable. We work in a model of separation where $\varphi * \psi$ additionally implies probabilistic independence of the events described by φ and ψ. The idea to model independence using separation was first explored by Barthe et al. [2020] in Probabilistic Separation Logic (PSL). In Chapter 3, we give a new model of probabilistic separation that supports more powerful forms of case analysis compared to prior work. Case analysis is a crucial component of reasoning about probabilistic programs, as we will see in the next section.

1.2.2 Case Analysis and Conditioning

When the execution of a probabilistic program branches into many possible random outcomes, it is often necessary to do case analysis, or *condition*, on those outcomes. In the concurrent shuffling algorithm from Figure 1.1, we already got a glimpse of how PCOL enables us to derive a compositional specification about concurrently executing threads:

$$\langle \lceil a_1 \mapsto A_1 \rceil * \lceil a_2 \mapsto A_2 \rceil \rangle \; \texttt{Shuffle}_1 \parallel \texttt{Shuffle}_2 \; \langle a_1 \sim \mathbf{unif}\,(\Pi(A_1)) * a_2 \sim \mathbf{unif}\,(\Pi(A_2))) \rangle$$

But the specification above only pertains to a single outcome of the **Partition** procedure, which has 2^n uniformly random outcomes where n is the length of the input list. We can therefore view

$a_1 \sim \mathbf{unif}\,(\Pi(A_1)) * a_2 \sim \mathbf{unif}\,(\Pi(A_2)))$ not as giving the actual probabilities that a_1 and a_2 take on particular values, but rather the *conditional probability* given that the input list ℓ has been partitioned into A_1 and A_2. In PCOL, reasoning about conditional probabilities is achieved using the SPLIT rule, which is inherited from standard Outcome Logic [Zilberstein 2025].

$$\frac{\langle \varphi \rangle\ C\ \langle \psi \rangle}{\langle \bigoplus_{X \sim d} \varphi \rangle\ C\ \langle \bigoplus_{X \sim d} \psi \rangle}\textsc{Split}$$

SPLIT essentially states that if the random outcomes of the program are initially distributed according to d, then we can condition on d and reason about the program as if X has a fixed value, before putting the postcondition back under an outcome conjunction. As we will see in Chapter 4, splitting enables a whole host of analysis applications. On top of enabling us to reason about the concurrent execution in **`ConcurrentShuffle`**, splitting is also required to handle control flow. For example, the program below branches based on a random coin flip.

$$x \sim \mathbf{Ber}\left(\tfrac{1}{2}\right) \fatsemi \mathtt{if}\ x = 0\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2$$

SPLIT allows us to condition on whether the guard of the if statement evaluates to true or false, and thus we can determine the full distribution of program outcomes because we will know whether C_1 or C_2 runs in each case. Prior variants of PSL had either no splitting rules [Li et al. 2023] or more limited ones [Bao et al. 2025], making PCOL a more powerful logic for reasoning about probabilistic control flow.

1.2.3 Shared State and Nondeterminism

We have now seen a brief sketch of the logical capabilities needed to prove correctness of the **`ConcurrentShuffle`** program. While already involved, that program does not have any shared state, and so the result is essentially deterministic since the interleaving order does not matter. Even larger challenges arise when the threads do have shared state, such as in the program below.

$$\left(z := 1 \fatsemi x :\approx \mathbf{Ber}\left(\tfrac{1}{2}\right) \fatsemi z := x\right) \quad \Big\| \quad y :\approx \mathbf{Ber}\left(\tfrac{z}{2}\right)$$

In the schedule where the entire left thread is executed before the right one, the program is effectively equivalent to $x :\approx \mathbf{Ber}\left(\frac{1}{2}\right) ⨟ y :\approx \mathbf{Ber}\left(\frac{x}{2}\right)$, so x and y are clearly correlated since the probability of y being 0 or 1 is directly influenced by the value of x. Therefore, the pcOL rule for composing the specifications for two concurrent threads (Par) has additional simple side conditions to ensure that the scheduler cannot induce an observable correlation between the distributions of outcomes in the two threads.

Achieving that *observable independence* requires several pieces of complex machinery. In Chapter 3, we see how the assertion language of preconditions and postconditions allows us to talk about independence at varying levels of granularity. Later, in Chapters 5 and 7, we present sound strategies for reasoning about nondeterministic outcomes that do not violate the probabilistic independence guarantees of pcOL.

1.2.4 Semantics and Soundness

The previous sections have provided a brief tour of the reasoning principles of pcOL. Those expressive forms of reasoning are only possible due to a new semantic model for probabilistic concurrent programs, which is able to model the entire set of distributions resulting from all the possible interleavings of parallel probabilistic threads. That new model, called *pomsets with formulae*, handles loops using domain-theoretic foundations, which provide compositional ways to approximate the behavior of unbounded computations using finite unrollings.

The key difficulty in modeling concurrent programs is that the semantics is not compositional. As demonstrated by the program below, executing two programs in parallel can produce behaviors that are not observable by an isolated execution of either thread.

$$x \coloneqq 0 ⨟ \mathbf{if}\ x = 0\ \mathbf{then}\ y \coloneqq 0\ \mathbf{else}\ y \coloneqq 1 \ \Big\|\ x \coloneqq 1$$

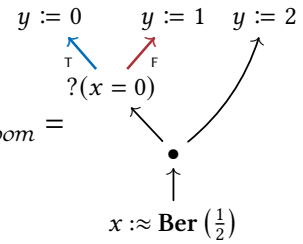
Here, the left thread will always assign y to 0 and the right thread does not assign y at all, but when composed in parallel, there is an interleaving where y gets assigned to 1. Because of that, the semantics of parallel programs cannot simply be given as a state transformer function that

takes in start states and returns end states, as there would be no way to determine the semantics of $C_1 \parallel C_2$ from the semantics of C_1 and C_2.

Instead, the dominant denotational technique for concurrency semantics, partially ordered multisets (pomsets) [Gischer 1988; Pratt 1976], models the causality between the actions in a program using a partial order. However, prior variants of pomsets could not properly model the randomized control flow needed to handle probabilistic programs. By contrast, pomsets with formulae introduce a new type of guarded branching, which allows all of the random outcomes to be represented within a single semantic structure. For example, the program below on the left corresponds to the pomset with formulae on the right.

$$C \triangleq x :\approx \mathbf{Ber}\left(\tfrac{1}{2}\right) \fatsemi \left(\begin{array}{l} \textbf{if } x = 0 \textbf{ then} \\ \quad y := 0 \\ \textbf{else} \\ \quad y := 1 \end{array} \;\middle\|\; y := 2 \right)$$

$[\![C]\!]_{pom} =$



After the initial sampling operation, the program execution branches into two concurrent threads. That is modeled by the fact that the • node has two outgoing causality arrows, meaning that the commands that follow could be interleaved in any order. The later if statement results in different kinds causality arrows, which are only enabled if the test at their root passes or fails, respectively. The full details of pomsets with formulae are given in Chapter 6 and the soundness of pcOL with respect to that model is proven in Chapter 7.

1.3 OUTLINE AND PREVIOUSLY PUBLISHED WORK

As outlined in the prior section, the primary contribution of this thesis is the new pcOL program logic, which we build up iteratively from first principles. Because the theory of pcOL is so complex, it is best communicated in steps starting from pure probabilistic reasoning (Chapters 3 and 4), then adding nondeterminism (Chapter 5), then moving to a setting where nondeterminism arises from concurrent scheduling (Chapters 6 and 7).

That progression is also inspired by a sequence of six peer-reviewed papers published by the author and collaborators, introduced in Section 1.3.1. However, this thesis is not merely a concatenation of those papers. As described in Section 1.3.2, the technical content of the earlier papers has been reworked in order to incorporate later advancements, turning the entire body of work into a coherent development with consistent definitions and notation. A full outline of the chapters is given in Section 1.3.3.

1.3.1 Previously Published Work

The technical contents of this thesis draw on the six peer-reviewed publications described below. The author of this thesis is the first author and lead contributor to all of the works.

Outcome Logic. Outcome Logic is a unified theory for reasoning about programs with different kinds of *computational effects* such as nondeterminism, randomization, and exceptions. It was first developed as a means to reason about both correctness (verification) and incorrectness (bug-finding) within a single logic through two papers published at OOPSLA 2023 and 2024.

> Noam Zilberstein, Derek Dreyer, and Alexandra Silva. 2023. Outcome Logic: A Unifying Foundation for Correctness and Incorrectness Reasoning. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 93 (Apr 2023), 29 pages. https://doi.org/10.1145/3586045
>
> Noam Zilberstein, Angelina Saliling, and Alexandra Silva. 2024. Outcome Separation Logic: Local Reasoning for Correctness and Incorrectness with Computational Effects. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 104 (Apr 2024), 29 pages. https://doi.org/10.1145/3649821

Later, the core foundations were refined and generalized in the following TOPLAS 2025 paper, which focused more on the proof theory and extended the rules to handle unbounded loops.

> Noam Zilberstein. 2025. Outcome Logic: A Unified Approach to the Metatheory of Program Logics with Branching Effects. *ACM Trans. Program. Lang. Syst.* Vol 47, Issue 3, Article 14 (Sept 2025), 71 pages. https://doi.org/10.1145/3743131

Demonic Outcome Logic (dOL). Whereas standard Outcome Logic can be instantiated to reason about nondeterminism *or* randomization, dOL, introduced in the following POPL 2025 paper, handles the interaction between both effects in a single program.

Noam Zilberstein, Dexter Kozen, Alexandra Silva, and Joseph Tassarotti. 2025b. A Demonic Outcome Logic for Randomized Nondeterminism. *Proc. ACM Program. Lang.* 8, POPL, Article 19 (Jan 2025), 30 pages. https://doi.org/10.1145/3704855

The rules for reasoning about random outcomes in dOL carry over exactly from standard Outcome Logic, whereas nondeterminism is treated *demonically*. As such, a dOL specification states that the random outcomes of a program are distributed in a particular way regardless of how the adversarial nondeterminism is resolved.

Pomsets with Formulae. Most prior logics that combine randomization and concurrency have used operational models. The shortcoming of operational approaches is that they cannot account for the infinitary aggregation of all the individual steps in order to produce the entire set of distributions that result from both nondeterministic and probabilistic computation. The following paper from CONCUR 2025 introduced pomsets with formulae, the first semantic model for programs that are both probabilistic and concurrent with unbounded while loops.

Noam Zilberstein, Daniele Gorla, and Alexandra Silva. 2025a. Denotational Semantics for Probabilistic and Concurrent Programs. In *36th International Conference on Concurrency Theory (CONCUR 2025) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 348)*, Patricia Bouyer and Jaco van de Pol (Eds.). Schloss Dagstuhl – Leibniz-Zentrum f"ur Informatik, Dagstuhl, Germany, 39:1–39:24. https://doi.org/10.4230/LIPIcs.CONCUR.2025.39

Pomsets with formulae are equipped with a linearization operator, which results in the exact same semantics as was used in dOL when specialized to programs without any concurrent execution. The connection between the two models laid the groundwork for a concurrent extension of dOL.

Probabilistic Concurrent Outcome Logic (pcOL). In a culmination of the five prior papers, the following paper, which won a Distinguished Paper Award at POPL 2026, introduced pcOL.

Noam Zilberstein, Alexandra Silva, and Joseph Tassarotti. 2026. Probabilistic Concurrent Reasoning in Outcome Logic: Independence, Conditioning, and Invariants. *Proc. ACM Program. Lang.* 10, POPL (Jan 2026), 30 pages. https://doi.org/10.1145/3776651

As we saw in Section 1.2, pcOL uses probabilistic independence to construct the joint distribution over all the variables in concurrent threads. That required a new measure-theoretic assertion

model to simultaneously reason about probabilistic and nondeterministic outcomes à la DOL, and also expressions about independence at varying granularity. As a result, PCOL is the first logic that combines expressive probabilistic reasoning with compositional concurrency reasoning.

1.3.2 Changes from Prior Work

Although the six papers described in Section 1.3.1 already constitute a coherent line of work, significant advancements were made later in the development, which informed improvements to technical content of the earlier papers. As such, the presentation in this thesis has been thoroughly reworked by interspersing the latest insights throughout the entire progression.

In particular, all of the logics in this thesis are based on the measure-theoretic assertion model of PCOL, which was introduced by Zilberstein et al. [2026]. Some of the challenges and insights related to designing PCOL around that model therefore appear in this thesis before turning to concurrency. For example, the interaction between probabilistic independence and control flow shows up when reasoning about purely probabilistic programs (Chapter 4), and the challenges around controlling for correlations induced by the scheduler show up in DOL (Chapter 5).

In addition, new results formally show how the logics introduced in each chapter build on each other. Theorem 5.8 shows that DOL is a *conservative extension* of Probabilistic Outcome Logic (POL). Formally, that means that for any program C, which is syntactically limited to only use probabilistic (and not nondeterministic) constructs, a DOL specification about C has the exact same meaning as a POL specification about C. Similarly, Theorem 7.4 and Corollary 7.5 show that PCOL is a conservative extension of DOL and POL, respectively. As a result, the soundness of all three logics stem from a single correctness proof in Theorem 7.10.

Although PCOL itself is essentially unchanged from the version due to Zilberstein et al. [2026], the presentation in this thesis is entirely self contained. As such, this thesis provides a more comprehensive reference on the design choices, semantic models, and historical context that contributed to the final logic.

1.3.3 Outline

The chapters of this thesis form a progression in which the entire pcOL theory is iteratively built up from first principles. As outlined below, each chapter builds on the previous ones to introduce more capabilities into the logical approach, while also highlighting new insights and challenges. Chapters 3 to 7 each begin with an overview section outlining the key ideas. Chapters 4, 5 and 7, which introduce pOL, dOL, and pcOL, use two running examples to demonstrate how the capabilities of each logic builds on the previous iteration.

Chapter 2: Preliminaries and Mathematical Foundations. The technical chapters begin by providing the mathematical background that will underlie the remainder of the development. We start with a recipe for defining denotational semantics, and show how that recipe can be instantiated to model probabilistic programs. We also introduce a variety of program logics that served as inspiration for pcOL.

Chapter 3: A Measure Theoretic Model of Outcomes. In the first technical contribution, we introduce a language of *outcome assertions* for expressing properties about the distribution of outcomes that can occur from running a probabilistic program. Outcome assertions include both the ability quantify the probabilities of outcomes and also to reason modularly about distributions over subsets of the program variables. The technical content is based on Zilberstein et al. [2026].

Chapter 4: Probabilistic Outcome Logic. Probabilistic Outcome Logic (pOL) is the first iteration of our program logic. It has expressive inference rules for reasoning about the distributions of outcomes in purely probabilistic programs with unbounded loops. The technical content is based on Zilberstein [2025] and Zilberstein et al. [2026].

Chapter 5: Demonic Outcome Logic. Demonic Outcome Logic (dOL) extends pOL for reasoning about probabilistic programs, which additionally feature a demonic adversary that con-

trols nondeterministic choices. Modeling the interaction between random and demonic choice requires a more sophisticated semantic model, and reasoning about that interaction introduces new challenges. For example, dOL must carefully control for probabilistic correlations between local and global state that could be caused by demonically controlled nondeterminism. The technical content is based on Zilberstein et al. [2025b, 2026].

Chapter 6: Denotational Semantics for Probabilistic Concurrent Programs. In a detour from program logics, we define the pomsets with formulae semantic model, which we introduced at the beginning of Section 1.2. Pomsets with formulae are carefully designed to be compatible with the convex powerdomain, which underlies the soundness of dOL. The technical content is based on Zilberstein et al. [2025a].

Chapter 7: Probabilistic Concurrent Outcome Logic. As a culmination of all the prior chapters, pcOL extends dOL for expressive reasoning about randomized concurrency, with metatheory backed by pomsets with formulae. The logic uses separation to compositionally reason about threads in a concurrent programs and resource invariants to handle shared state. The nondeterministic reasoning abilities of dOL prove crucial when inspecting shared state that may be altered by parallel threads. The technical content is based on Zilberstein et al. [2026].

Chapter 8: Case Studies. To demonstrate the cumulative features of the prior chapters, we present correctness proofs for three more involved example programs including the concurrent shuffling algorithm from Section 1.2. The examples first appeared in Zilberstein et al. [2026].

Chapter 9: Conclusion. We conclude by discussing applications of pcOL and future work to support larger scale verification efforts. That future work, some of which is already underway, includes both further theoretical developments and proposed applications of the logic.

CHAPTER 2

PRELIMINARIES AND MATHEMATICAL FOUNDATIONS

Formal methods are mathematically rigorous techniques for proving properties about the behavior of programs. Those techniques rely on establishing a precise semantics or *meaning* for programs, which is phrased as a complex mathematical object. The complexity increases further with the addition of *computational effects* such as randomization, concurrency, and unbounded looping. To tame that complexity, we often do not need to specify the program's precise meaning, but rather only a *sound approximation* of it.

Although there are many techniques for specifying the behavior of a program, this thesis focuses on program logics. Program logics like Hoare Logic [Hoare 1969] provide a language to describe the behavior of a program in terms of preconditions and postconditions. In classical Hoare Logic, pre- and postconditions describe individual program states, but the inclusion of effects such as randomization means that they must describe more complex properties, such as how the program outcomes are randomly distributed.

In this chapter, we survey the mathematical concepts that underlie the development of program logics. First, we discuss an extensible strategy for defining program semantics in Section 2.1. Next, we instantiate that approach to model probabilistic programs in Section 2.2. We then introduce Hoare Logic (Section 2.3), separation logic (Section 2.4), and the author's prior work on Outcome Logic (Section 2.5), all of which inspired the contributions in later chapters.

2.1 DENOTATIONAL SEMANTICS AND DOMAIN THEORY

Program semantics has been studied in many styles and for many different types of languages. In this dissertation, we will work with imperative languages, stemming from a basic syntax, shown in Figure 2.1, which includes no-ops (**skip**); sequential composition ($C_1 \fatsemi C_2$); conditional branching (**if** b **then** C_1 **else** C_2), where b is a Boolean test; while loops (**while** b **do** C); and basic actions $a \in act$. We will later add more constructs, such as probabilistic choice $C_1 \oplus_p C_2$,

$$
\begin{array}{lll}
\mathit{cmd} \ni C ::= & \texttt{skip} & \textit{(No-op)} \\
& \mid C_1 \fatsemi C_2 & \textit{(Sequential Composition)} \\
& \mid \texttt{if } b \texttt{ then } C_1 \texttt{ else } C_2 & \textit{(Conditional Branching)} \\
& \mid \texttt{while } b \texttt{ do } C & \textit{(Loops)} \\
& \mid a & \textit{(Basic Actions } a \in \mathit{act}\textit{)}
\end{array}
$$

Figure 2.1: Syntax of Command Language (CL) programs, where $a \in \mathit{act}$.

nondeterministic choice $C_1 \,\&\, C_2$, and parallel execution $C_1 \parallel C_2$, but the preliminary language in Figure 2.1 already demonstrates core challenges.

The two dominant approaches to program semantics are *operational* and *denotational*. In an operational model, the program is modeled as a sequence of steps, and the final outcomes consist of all states that can be reached via a finite number of steps, plus nontermination if any infinite sequences of steps exist. Operational models are often preferred for concurrency, as it is easy to model interleaving behavior by simply stating that any thread may step at each point in time. However, operational models less readily model probabilistic programs, as operational techniques lack the tools to determine the *probability distribution* of outcomes in the limit.

For example, consider the following program, in which the loop increments i on each iteration and then exits with probability $\frac{1}{2}$. Note that $\mathbf{Ber}\,(p)$ is the *Bernoulli* distribution, which returns 1 with probability p and 0 with probability $1 - p$.

$$
\begin{array}{l}
x \coloneqq 0 \fatsemi \\
i \coloneqq 0 \fatsemi \\
\texttt{while } x = 0 \texttt{ do} \\
\quad x :\approx \mathbf{Ber}\left(\frac{1}{2}\right) \fatsemi \\
\quad i \coloneqq i + 1
\end{array}
$$

It is relatively easy to see that upon termination, i will be distributed according to a *geometric* distribution, where $i = n$ with probability $\frac{1}{2^n}$ for all $n \in \mathbb{N}$. However, there is no finite number of steps that the program can take after which the full distribution of outcomes is known, since the probability of termination only reaches 1 after an infinite amount of time. As a result, domain-

theoretic tools [Scott and Strachey 1971; Scott 1972] are needed to determine the program's full behavior *in the limit*, motivating a move away from the operational style.

Denotational models interpret programs as mathematical objects in some underlying domain. The semantics is a function $[\![-]\!] : \mathcal{S} \to T(\mathcal{S})$ mapping each state $s \in \mathcal{S}$ into a computational domain $T(\mathcal{S})$. The specific choice of T depends on the *computational effects* present in the language, for example, powersets encode the outcomes of nondeterministic programs whereas probability distributions encode the outcomes of probabilistic programs. We discuss the necessary properties of T throughout the remainder of this section.

2.1.1 Computational Effects and Actions

For our purposes, we consider computational effects to be any actions performed by a program other than returning a state. The command language outlined above already displays a computational effect; the inclusion of while loops opens up the possibility of *nontermination.* As a result, programs cannot simply be interpreted as maps from states to states, but rather they must be interpreted as maps from states to states or nontermination, $[\![-]\!] : \mathcal{S} \to \mathcal{S}_\perp$ where $\mathcal{S}_\perp = \mathcal{S} \cup \{\perp\}$ and $\perp$ represents nontermination.

Another common effect is nondeterminism, where the program may branch into many possible outcomes instead of just one. In that case, we interpret programs as maps from states to sets of states $[\![-]\!] : \mathcal{S} \to 2^{\mathcal{S}}$ where $2^{\mathcal{S}}$ is the powerset of $\mathcal{S}$. In the resulting *powerdomain*, nontermination is represented as $\emptyset$, the empty set of outcomes. However, we cannot distinguish programs that *always* terminate from ones that *sometimes* terminate. In general, nondeterminism has a very complex interaction with nontermination; the interested reader can refer to Apt and Plotkin [1986], Søndergaard and Sestoft [1992], and Back [1980, 1983] for more details.

The command language introduced in Figure 2.1 did not specify what actions $a \in act$ can be performed. The actions that can be represented in each instantiation of the semantic model depend on the computational domain that we choose to use. For example, in the powerset model we

could include nondeterministic assignment $x \leftarrow S$, where the variable x is nondeterministically assigned some value from the set S. To formalize the semantics of actions, we first must describe the programs states that they can manipulate.

Definition 2.1 (Memories)**.** *Let* Var *be a countable set of variable names and* Val *be the set of program values. Then for any* $V \subseteq \mathsf{Var}$*, the set of memories over* V *is defined as the set of maps from* V *to values.*

$$\mathsf{Mem}[V] \triangleq V \to \mathsf{Val}$$

For any $x \in V$ *and* $v \in \mathsf{Val}$*, a memory* $\sigma \in \mathsf{Mem}[V]$ *can be updated as follows:*

$$\sigma[x := v](y) \triangleq \begin{cases} v & \text{if } x = y \\ \sigma(x) & \text{if } x \neq y \end{cases}$$

Given the above definition of memories, and a computational domain T (which could be the nontermination or nondeterminism domains that we already saw), an action can be any operation whose semantics is representable via a function of type $[\![-]\!]_{act} : \mathsf{Mem}[V] \to T(\mathsf{Mem}[V])$. For now, we will only include a single action: variable assignment $x := e$. To do so, we first need a notion of expressions, which include variables $x \in \mathsf{Var}$, values $v \in \mathsf{Val}$, and a variety of different arithmetic expressions.

$$exp \ni e ::= x \mid v \mid e_1 + e_2 \mid e_1 - e_2 \mid \cdots$$

Expressions are evaluated using the semantic function $[\![-]\!]_{exp} : exp \to \mathsf{Mem}[V] \to \mathsf{Val}$, which is defined in the standard way, as long as the free variables of the expression are contained in the given memory. Given that, we can define the semantics of variable assignment in the nontermination domain $[\![x := e]\!]_{act} : \mathsf{Mem}[V] \to \mathsf{Mem}[V]_\perp$ as follows:

$$[\![x := e]\!]_{act}(\sigma) \triangleq \begin{cases} \sigma[x := [\![e]\!]_{exp}(\sigma)] & \text{if } x \in V \text{ and } \mathsf{vars}(e) \subseteq V \\ \perp & \text{otherwise} \end{cases}$$

2.1.2 Sequential Composition and Monads

We have already seen two examples of computational domains for representing nonterminating computations and nondeterministic computations. However, those domains need additional

properties in order to ensure that we can define the semantics of all commands properly. In this section, we focus on sequential composition.

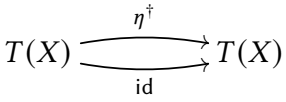
To do so, we will use *monads*, an established method of sequencing computational effects [Moggi 1991, 1989; Wadler 1993]. Presuming that we know how to interpret the semantics of $[\![C_1]\!] : \mathsf{Mem}[V] \to T(\mathsf{Mem}[V])$ and $[\![C_2]\!] : \mathsf{Mem}[V] \to T(\mathsf{Mem}[V])$, monads provide us with a *Kleisli extension* operation $(-)^\dagger$ to compose $[\![C_1]\!]$ and $[\![C_2]\!]$ into a semantic function capturing the sequential command $C_1 \fatsemi C_2$. More formally, we define monads as follows:

Definition 2.2 (Monads). *A Kleisli triple $\langle T, \eta, (-)^\dagger \rangle$ in* **Set** *consists of a functor T: **Set** $\to$ **Set**, and two morphisms:*

$$\eta : X \to T(X) \qquad\qquad (-)^\dagger : (X \to T(Y)) \to T(X) \to T(Y)$$

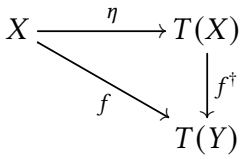
such that the following diagrams commute:

Left Identity



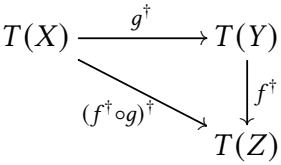
$\eta^\dagger = \mathsf{id}$

Right Identity



$f^\dagger \circ \eta = f$

Associativity



$f^\dagger \circ g^\dagger = (f^\dagger \circ g)^\dagger$

If such a Kleisli triple exists, then T is a monad.

Monads can capture a wide variety effects including nontermination, nondeterminism, exceptions, probabilistic computation, and certain combinations thereof. For a more thorough introduction, refer to Awodey [2006]. We now give examples of two such instantiations, for the effects introduced previously.

Example 1 (Nontermination). *The functor $\langle (-)_\perp, \eta, (-)^\dagger \rangle$ is a monad, where:*

$$\eta(x) \triangleq x \qquad\qquad f^\dagger(x) \triangleq \begin{cases} f(x) & \text{if } x \neq \perp \\ \perp & \text{if } x = \perp \end{cases}$$

Example 2 (Nondeterminism). *The powerset functor $\langle \mathbb{2}^{(-)}, \eta, (-)^\dagger \rangle$ is a monad, where:*

$$\eta(x) \triangleq \{x\} \qquad\qquad f^\dagger(S) \triangleq \bigcup_{x \in S} f(x)$$

With the above examples in mind, we can now define the semantics of **skip** and sequential composition. The **skip** command is simply a no-op, so it uses the monad unit η to return the current state. For sequential composition $C_1 \fatsemi C_2$, the Kleisli extension is applied to the second command to obtain $[\![C_2]\!]^\dagger : T(\mathsf{Mem}[V]) \to T(\mathsf{Mem}[V])$, which can then be applied after $[\![C_1]\!] : \mathsf{Mem}[V] \to T(\mathsf{Mem}[V])$ via regular function composition '$\circ$'.

$$[\![\textbf{skip}]\!] \triangleq \eta \qquad\qquad [\![C_1 \fatsemi C_2]\!] \triangleq [\![C_2]\!]^\dagger \circ [\![C_1]\!]$$

So, expanding the definitions, we get that the semantics of $C_1 \fatsemi C_2$ in the nontermination model is simply $[\![C_2]\!]$ executed after $[\![C_1]\!]$, if C_1 terminates otherwise it is $\bot$.

$$[\![C_1 \fatsemi C_2]\!](\sigma) = \begin{cases} [\![C_2]\!]([\![C_1]\!](\sigma)) & \text{if } [\![C_1]\!](\sigma) \neq \bot \\ \bot & \text{if } [\![C_1]\!](\sigma) = \bot \end{cases}$$

In the nondeterminism model, the semantics of $C_1 \fatsemi C_2$ is the union of all the outcomes of running C_2 in all the end states of C_1.

$$[\![C_1 \fatsemi C_2]\!](\sigma) = \bigcup_{\tau \in [\![C_1]\!](\sigma)} [\![C_2]\!](\tau)$$

In addition, the monad laws imply the following equational laws, stating that **skip** is the left and right identity of sequential composition, and that sequential composition is associative.

$$\textbf{skip} \fatsemi C \;\equiv\; C \fatsemi \textbf{skip} \;\equiv\; C \qquad\qquad (C_1 \fatsemi C_2) \fatsemi C_3 \;\equiv\; C_1 \fatsemi (C_2 \fatsemi C_3)$$

Above, we take $C \equiv C'$ to mean that $[\![C]\!](\sigma) = [\![C']\!](\sigma)$ for all $\sigma \in \mathsf{Mem}[V]$.

2.1.3 Tests and Conditional Branching

We next examine the deterministic branching construct **if** b **then** C_1 **else** C_2. If statements do not require any new properties of semantic domains, but we must define the *tests* that are used to determine control flow. Tests contain the constants **true** and **false**; they are closed under conjunctions, disjunction, and negations; and also include a variety of comparators.

$$\mathit{test} \ni b ::= \textbf{true} \mid \textbf{false} \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \mid \neg b \mid e_1 = e_2 \mid e_1 \leq e_2 \mid \cdots$$

The semantics of tests is standard, and is given by the function $[\![-]\!]_{test} : \mathit{test} \to \mathsf{Mem}[V] \to \mathbb{B}$, where $\mathbb{B} = \{0, 1\}$ is the set of Booleans with 1 representing true and 0 representing false.

Given the interpretation of tests, the semantics of if statements is equal to the semantics of either the true or false branch, depending on the evaluation of the test.

$$[\![\textbf{if } b \textbf{ then } C_1 \textbf{ else } C_2]\!]\,(\sigma) \coloneqq \begin{cases} [\![C_1]\!]\,(\sigma) & \text{if } [\![b]\!]_{test}\,(\sigma) = 1 \\ [\![C_2]\!]\,(\sigma) & \text{if } [\![b]\!]_{test}\,(\sigma) = 0 \end{cases}$$

Going forward, we will occasionally use the one-sided if statement '**if** b **then** C' as syntactic sugar for '**if** b **then** C **else skip**'.

2.1.4 Loops and Fixed Points

While loops can result in unbounded iteration, therefore it is not generally possible to fully determine a loop's behavior via finite unrolling. As a first attempt, we could define the semantics of while loops recursively as follows:

$$\begin{aligned} [\![\textbf{while } b \textbf{ do } C]\!]\,(\sigma) &= [\![\textbf{if } b \textbf{ then } (C \fatsemi \textbf{while } b \textbf{ do } C) \textbf{ else skip}]\!]\,(\sigma) \\ &= \begin{cases} [\![\textbf{while } b \textbf{ do } C]\!]^{\dagger}([\![C]\!]\,(\sigma)) & \text{if } [\![b]\!]_{test}\,(\sigma) = 1 \\ \eta(\sigma) & \text{if } [\![b]\!]_{test}\,(\sigma) = 0 \end{cases} \end{aligned}$$

The problem is that the recursion above is not well-founded. For example, in the case of the infinite loop '**while true do skip**', the above expression reduces to:

$$\begin{aligned} [\![\textbf{while true do skip}]\!]\,(\sigma) &= \begin{cases} [\![\textbf{while true do skip}]\!]^{\dagger}([\![\textbf{skip}]\!]\,(\sigma)) & \text{if } [\![\textbf{true}]\!]_{test}\,(\sigma) = 1 \\ \eta(\sigma) & \text{if } [\![\textbf{true}]\!]_{test}\,(\sigma) = 0 \end{cases} \\ &= [\![\textbf{while true do skip}]\!]\,(\sigma) \end{aligned}$$

But that does not give any insight into the program's meaning. So, to make the semantics formal, we need to find some function f such that for all $\sigma \in \mathsf{Mem}[V]$:

$$f(\sigma) = \begin{cases} f^{\dagger}([\![C]\!]\,(\sigma)) & \text{if } [\![b]\!]_{test}\,(\sigma) = 1 \\ \eta(\sigma) & \text{if } [\![b]\!]_{test}\,(\sigma) = 0 \end{cases}$$

The key to finding f lies in fixed point theory. A fixed point of a function $g\colon X \to X$ is an element $x \in X$ such that $g(x) = x$. So, f above is a *fixed point* of the *loop characteristic function* $\Phi_{\langle b,C\rangle}$.

$$\Phi_{\langle b,C\rangle}(f)(\sigma) \triangleq \begin{cases} f^{\dagger}(\llbracket C \rrbracket(\sigma)) & \text{if } \llbracket b \rrbracket_{test}(\sigma) = 1 \\ \eta(\sigma) & \text{if } \llbracket b \rrbracket_{test}(\sigma) = 0 \end{cases}$$

Proving the existence of such fixed points can be done using domain theory, which we introduce briefly now. For a more comprehensive introduction, refer to Abramsky and Jung [1995]. Domain theory will tell us what additional properties of $T(\mathsf{Mem}[V])$ are needed in order to ensure that the semantics of loops is well-defined. More precisely, we will formalize the notion of approximating loops via an order $\leq$, and prove that each successive unrolling of a loop gives us a larger approximation until we eventually converge to the real meaning.

Definition 2.3 (Partially Ordered Set)**.** *A partially ordered set, or poset, $\langle X, \leq\rangle$ is a set X equipped with a relation $\leq \subseteq X \times X$ obeying the following properties:*

- ***Reflexivity**: $\forall x \in X, x \leq x$*
- ***Transitivity**: $\forall x, y, z \in X, x \leq y \wedge y \leq z \Rightarrow x \leq z$*
- ***Antisymmetry**: $\forall x, y \in X, x \leq y \wedge y \leq x \Rightarrow x = y$*

*When the relation is irreflexive instead of reflexive (*i.e., *$x \not\leq x$), we use the symbol $<$ instead of $\leq$ and say that the poset $\langle X, <\rangle$ is strict.*

Posets abstract the notion of accumulating terminating outcomes of a loop after each successive finite approximation. For example, the nontermination domain $\langle \mathsf{Mem}[V]_{\perp}, \sqsubseteq_{\mathsf{flat}}\rangle$ is a poset, where $\sqsubseteq_{\mathsf{flat}}$ is the *flat* order, $\perp$ is the smallest element, and all pairs of unequal states are not comparable to each other:

$$\sigma \sqsubseteq_{\mathsf{flat}} \tau \qquad \text{iff} \qquad \sigma = \perp \quad \text{or} \quad \sigma = \tau$$

In the flat order, the finite approximations of the loop are $\perp$, until the loop terminates in some state τ. Clearly $\perp \sqsubseteq_{\mathsf{flat}} \tau$, so the semantics gets larger as we consider more and more approximants.

The nondeterminism domain $\left\langle 2^{\mathsf{Mem}[V]}, \subseteq \right\rangle$ is also a poset, ordered by standard subset inclusion. Here, each iteration of the loop may produce more terminating outcomes, so we get a larger and larger set as we consider more approximants.

Next, we need to ensure that these approximants converge to some well-defined value. We will do that by taking the supremum, or least upper bound, of the sequence.

Definition 2.4 (Supremum)**.** *Given a poset $\langle X, \leq \rangle$ and a set $A \subseteq X$, the supremum* $\sup A$ *(also called least upper bound) is an element $x \in X$ such that $y \leq x$ for all $y \in A$ and $x \leq z$ for any $z \in X$ such that $y \leq z$ for all $y \in A$.*

Not all sequences have a supremum, which is why the tools of domain theory play a key role in ensuring that a program semantic is well-defined. In our case, we will specifically be interested in suprema of *directed sets*.

Definition 2.5 (Directed Set)**.** *Given a poset $\langle X, \leq \rangle$, a set $D \subseteq X$ is* directed *if it is nonempty and every pair of elements in D has an upper bound in D. That is, for all $x, y \in D$ there exists $z \in D$ such that $x \leq z$ and $y \leq z$.*

Note that an ascending chain $x_1 \leq x_2 \leq x_2 \leq \cdots$ is a special kind of directed set. Now, to ensure that the semantics of loops converge, we will restrict to domains where every directed set has a supremum. These domains are called *directed complete partial orders*.

Definition 2.6 (Directed Complete Partial Order)**.** *A directed complete partial order, or* DCPO*, is a poset $\langle X, \leq \rangle$ such that every directed set $D \subseteq X$ has a supremum. If there is a bottom element $\bot \in X$ such that $\bot \leq x$ for all $x \in X$, then $\langle X, \leq \rangle$ is a* pointed DCPO*.*

Once again, both the nontermination and nondeterminism domains are DCPOs. In the nontermination domain, every directed set has the form $\{\bot\}$, $\{\sigma\}$, or $\{\bot, \sigma\}$; it is not possible for a directed set to contain two distinct elements σ and τ, since they have no upper bound according to $\sqsubseteq_{\mathsf{flat}}$. It is easy to see that $\sup\{\bot\} = \bot$ and $\sup\{\sigma\} = \sup\{\bot, \sigma\} = \sigma$. In the case of nondeterminism, suprema are given by set union.

With the above domain theoretic machinery in place, the final step towards defining the semantics of while loops is to formulate that semantics via fixed points. To that end, we introduce the notion of Scott continuity, and a well known result stating that all Scott continuous functions have a least fixed point.

Definition 2.7 (Scott Continuity)**.** *Given two* DCPOs $\langle X, \leq_X \rangle$ *and* $\langle Y, \leq_Y \rangle$*, a function* $f : X \to Y$ *is Scott Continuous if it is monotone:*

$$\forall x, x' \in X. \quad x \leq_X x' \implies f(x) \leq_Y f(x')$$

and preserves suprema of any directed set $D \subseteq X$*:*

$$\sup_{x \in D} f(x) = f(\sup D)$$

Theorem 2.8 (The Kleene Fixed Point Theorem)**.** *Let* $\langle X, \leq \rangle$ *be a pointed* DCPO *and* $f : X \to X$ *be a Scott continuous function. Then* f *has a least fixed point given by:*

$$\mathsf{lfp}(f) = \sup_{n \in \mathbb{N}} f^n(\bot)$$

where $f^0 = \mathsf{id}$ *and* $f^{n+1} = f \circ f^n$*.*

So, the characteristic function $\Phi_{\langle b,C \rangle}$ has a least fixed point as long as it is Scott continuous and the underlying domain $\langle T(\mathsf{Mem}[V]), \sqsubseteq \rangle$ is a DCPO. In fact, it is sufficient for the Kleisli extension to be Scott continuous in its first argument.

Lemma 2.9. *Let* $\langle T(\mathsf{Mem}[V]), \sqsubseteq \rangle$ *be a* DCPO *and* $\left\langle T, \eta, (-)^\dagger, \right\rangle$ *be a monad. Let* $\sqsubseteq^\bullet$ *be the pointwise extension of* $\sqsubseteq$*,* i.e.*,* $f \sqsubseteq^\bullet g$ *iff* $f(x) \sqsubseteq g(x)$ *for all* x*. If the Kleisli extension is Scott continuous,* i.e.*, for any* $x, y \in T(\mathsf{Mem}[V])$ *and directed set* $D \subseteq T(\mathsf{Mem}[V])$*:*

$$x \sqsubseteq y \implies f^\dagger(x) \sqsubseteq f^\dagger(y) \qquad\qquad f^\dagger(\sup D) = \sup_{x \in D} f^\dagger(x)$$

then the loop characteristic function $\Phi_{\langle b,C \rangle}$ *is also Scott continuous.*

Proof. The proof follows the same structure as Lemma B.12 of Zilberstein et al. [2025b]. □

Following from Lemma 2.9, $\Phi_{\langle b,C\rangle}$ has a least fixed point, and we can therefore use its least fixed point to give semantics to loops.

$$[\![\textbf{while } b \textbf{ do } C]\!] \triangleq \mathsf{lfp}\left(\Phi_{\langle b,C\rangle}\right)$$

By the Kleene Fixed Point Theorem, that fixed point is equal to the supremum of all finite iterates.

$$\mathsf{lfp}\left(\Phi_{\langle b,C\rangle}\right) = \sup_{n\in\mathbb{N}} \Phi^n_{\langle b,C\rangle}(\bot^\bullet)$$

Where $\bot^\bullet$ in the $\sqsubseteq^\bullet$ order is the function $\bot^\bullet(x) = \bot$. For any $n \in \mathbb{N}$, we also get that:

$$\Phi^{n+1}_{\langle b,C\rangle}(\bot) = [\![(\textbf{if } b \textbf{ then } C)^n]\!]$$

Where $C^0 = \textbf{skip}$ and $C^{n+1} = C \fatsemi C^n$. Therefore, the semantics of loops can be equivalently phrased as an infinite sequence of if statements:

$$[\![\textbf{while } b \textbf{ do } C]\!] = \sup_{n\in\mathbb{N}} [\![(\textbf{if } b \textbf{ then } C)^n]\!]$$

2.1.5 A Full Semantics

To summarize, we now give the full semantics for CL programs $C \in \mathit{cmd}$ from Figure 2.1. In order to interpret the language, we need to following components:

1. A domain T and set of states $\mathcal{S}$ such that $\langle T(\mathcal{S}), \sqsubseteq\rangle$ is a pointed DCPO.
2. A Kleisli triple $\left\langle T, \eta, (-)^\dagger\right\rangle$, such that the Kleisli extension $(-)^\dagger$ is Scott continuous.
3. Sets of actions act and tests test and semantic functions for actions and tests:

$$[\![-]\!]_{\mathit{act}} : \mathit{act} \to \mathcal{S} \to T(\mathcal{S}) \qquad\qquad [\![-]\!]_{\mathit{test}} : \mathit{test} \to \mathcal{S} \to \mathbb{B}$$

Given those components, the semantics is shown in Figure 2.2, where each component in thoroughly explained in one of the prior subsections.

As we will see throughout this thesis, the semantics presented in Figure 2.2 is remarkably extensible. We have already seen how it captures nonterminating and nondeterministic programs. Next, in Section 2.2, we will see how it also captures probabilistic computation, and later in Section 5.2 we will see how it captures computations that are both nondeterministic and probabilistic.

$$
\begin{aligned}
[\![\texttt{skip}]\!]\,(\sigma) &\triangleq \eta(\sigma) \\
[\![C_1 \mathbin{;} C_2]\!]\,(\sigma) &\triangleq [\![C_2]\!]^{\dagger}([\![C_1]\!]\,(\sigma)) \\
[\![\texttt{if}\ b\ \texttt{then}\ C_1\ \texttt{else}\ C_2]\!]\,(\sigma) &\triangleq \begin{cases} [\![C_1]\!]\,(\sigma) & \text{if } [\![b]\!]_{test}\,(\sigma) = 1 \\ [\![C_2]\!]\,(\sigma) & \text{if } [\![b]\!]_{test}\,(\sigma) = 0 \end{cases} \\
[\![\texttt{while}\ b\ \texttt{do}\ C]\!]\,(\sigma) &\triangleq \mathsf{lfp}\left(\Phi_{\langle C,b\rangle}\right)(\sigma) \\
[\![x \coloneqq e]\!]\,(\sigma) &\triangleq \eta(\sigma[x \coloneqq [\![e]\!]_{exp}\,(\sigma)])
\end{aligned}
$$

Figure 2.2: Denotational semantics of basic CL programs.

2.2 PROBABILISTIC PROGRAMMING

The study of the semantics of probabilistic programs dates back to the seminal work of Kozen [1979] and McIver and Morgan [2005]. In addition to the standard control flow operations from Figure 2.1, probabilistic programs can also perform random operations such as $C_1 \oplus_p C_2$, which executes C_1 with probability p and C_2 with probability $1 - p$; or $x :\approx d$, which samples a value from the discrete probability distribution d and stores the result in x. The full syntax is shown in Figure 2.3 and can be used to define a wide variety of randomized algorithms.

2.2.1 Preliminaries

The basic primitive for reasoning about probabilistic computation is the discrete probability distribution. A discrete distribution μ assigns probabilities to each element x from some carrier set X. It is also required that the cumulative probability of all the elements of X sums to 1.

Definition 2.10 (Discrete Probability Distribution)**.** *A discrete probability distribution is a countable map $\mu\colon X \to [0,1]$ from elements of a carrier set X to real-valued probabilities such that $\sum_{x\in X} \mu(x) = 1$. We let $\mathcal{D}(X)$ be the set of all discrete distributions over X.*

$$
\mathcal{D}(X) \triangleq \left\{ \mu\colon X \to [0,1] \;\middle|\; \sum_{x\in X} \mu(x) = 1 \right\}
$$

$$
\begin{array}{lll}
cmd_{\mathrm{P}} \ni C ::= \mathbf{skip} & & \textit{(No-op)} \\
\quad \mid C_1 \fatsemi C_2 & & \textit{(Sequential Composition)} \\
\quad \mid C_1 \oplus_e C_2 & & \textit{(Probabilistic Choice)} \\
\quad \mid \mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2 & & \textit{(Conditional Branching)} \\
\quad \mid \mathbf{while}\ b\ \mathbf{do}\ C & & \textit{(Loops)} \\
\quad \mid a & & \textit{(Basic Actions } a \in act\textit{)} \\
\\
a ::= x \coloneqq e & & \textit{(Assignment)} \\
\quad \mid x :\approx d(e) & & \textit{(Probabilistic Sampling)} \\
\\
d(\cdot) ::= \mathbf{Ber}\,(\cdot) \mid \mathbf{unif}\,(\cdot) \mid \mathbf{geo}\,(\cdot)
\end{array}
$$

Figure 2.3: Syntax of Probabilistic Command Language (pCL) programs.

The simplest example of a discrete probability distribution is the *Dirac* or *point-mass* distribution δ_x, which assigns probability 1 to x and 0 to everything else:

$$\delta_x(y) \triangleq \begin{cases} 1 & \text{if } x = y \\ 0 & \text{if } x \neq y \end{cases}$$

Additional examples are the Bernoulli distribution $\mathbf{Ber}\,(p) \in \mathcal{D}(\mathbb{B})$ which assigns probability p to 1 and $1 - p$ to 0; the uniform distribution $\mathbf{unif}\,(S) \in \mathcal{D}(S)$ over a finite set S, which assigns probability $\frac{1}{|S|}$ to each element $x \in S$; and the geometric distribution $\mathbf{geo}\,(p) \in \mathcal{D}(\mathbb{N})$, which assigns probability $p^n(1-p)$ to each natural number $n \in \mathbb{N}$.

$$\mathbf{Ber}\,(p)\,(x) \triangleq \begin{cases} p & \text{if } x = 1 \\ 1-p & \text{if } x = 0 \end{cases} \qquad \mathbf{unif}\,(S)\,(x) \triangleq \frac{1}{|S|} \qquad \mathbf{geo}\,(p)\,(n) \triangleq p^n(1-p)$$

The *support* of a distribution μ is the set of elements to which μ assigns nonzero probability.

$$\mathrm{supp}(\mu) \triangleq \{x \in X \mid \mu(x) > 0\}$$

So, for example, $\mathrm{supp}(\delta_x) = \{x\}$, $\mathrm{supp}(\mathbf{Ber}\,(p)) = \mathbb{B}$ (if $0 < p < 1$), $\mathrm{supp}(\mathbf{unif}\,(S)) = S$, and $\mathrm{supp}(\mathbf{geo}\,(p)) = \mathbb{N}$ (if $0 < p < 1$). Distributions also have a convex combination operator $\mu \oplus_p \nu$, which weights μ by p and ν by $1 - p$.

$$(\mu \oplus_p \nu)(x) \triangleq p \cdot \mu(x) + (1-p) \cdot \nu(x)$$

$$
\begin{aligned}
[\![\mathtt{skip}]\!]_{\mathcal{D}}\,(\sigma) &\triangleq \eta(\sigma) \\
[\![C_1 \fatsemi C_2]\!]_{\mathcal{D}}\,(\sigma) &\triangleq [\![C_2]\!]^{\dagger}_{\mathcal{D}}\,([\![C_1]\!]_{\mathcal{D}}\,(\sigma)) \\
[\![C_1 \oplus_e C_2]\!]_{\mathcal{D}}\,(\sigma) &\triangleq [\![C_1]\!]_{\mathcal{D}}\,(\sigma) \oplus_{[\![e]\!]_{exp}(\sigma)} [\![C_2]\!]_{\mathcal{D}}\,(\sigma) \\
[\![\mathtt{if}\ b\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2]\!]_{\mathcal{D}}\,(\sigma) &\triangleq \begin{cases} [\![C_1]\!]_{\mathcal{D}}\,(\sigma) & \text{if } [\![b]\!]_{test}\,(\sigma) = 1 \\ [\![C_2]\!]_{\mathcal{D}}\,(\sigma) & \text{if } [\![b]\!]_{test}\,(\sigma) = 0 \end{cases} \\
[\![\mathtt{while}\ b\ \mathtt{do}\ C]\!]_{\mathcal{D}}\,(\sigma) &\triangleq \mathsf{lfp}\left(\Phi_{\langle C,b\rangle}\right)(\sigma) \\
[\![x := e]\!]_{\mathcal{D}}\,(\sigma) &\triangleq \eta\left(\sigma[x := [\![e]\!]_{exp}\,(\sigma)]\right) \\
[\![x :\approx d(e)]\!]_{\mathcal{D}}\,(\sigma) &\triangleq \sum_{v \in \mathsf{Val}} d([\![e]\!]_{exp}\,(\sigma))(v) \cdot \eta\,(\sigma[x := v])
\end{aligned}
$$

Figure 2.4: Denotational semantics $[\![-]\!]_{\mathcal{D}} : cmd_{\mathrm{P}} \to \mathsf{Mem}[V] \to \mathcal{D}(\mathsf{Mem}[V]_{\perp})$ of the Probabilistic Command Language (PCL). The semantics of all commands except for the new probabilistic ones $C_1 \oplus_e C_2$ and $x :\approx d(e)$ carry over from Figure 2.2.

Now that we have defined discrete probability distributions, we can give semantics to PCL programs, using the following type.

$$[\![-]\!]_{\mathcal{D}} : cmd_{\mathrm{P}} \to \mathsf{Mem}[V] \to \mathcal{D}(\mathsf{Mem}[V]_{\perp})$$

The semantics of PCL is given in Figure 2.4, and is defined using the strategy outlined in Section 2.1 and Figure 2.2. That semantic model requires DCPO and monad structures for probability distributions, which we define in the following sections. We will also define semantics for the two new probabilistic command forms, $C_1 \oplus_e C_2$ and $x :\approx d(e)$.

2.2.2 The Probabilistic Powerdomain and Almost-Sure Termination

A powerdomain construction for probability distributions allows us to define the semantics of unbounded while loops for probabilistic programs. Whereas some probabilistic logics are based on operational techniques, those logics can only support bounded looping constructs because their underlying metatheories do not develop the domain theory needed to ensure that infinite computations are well-defined [Tassarotti and Harper 2019; Fan et al. 2025].

In that sense, probabilistic programs are fundamentally different from nondeterministic ones; in a nondeterministic program, one can quantify over all the finite traces in order to ensure that some safety property always holds. But probabilistic programs can have infinite traces whose probability only converges to 0 after an infinite amount of time, so reasoning about those programs requires more sophisticated mathematical tools. For example, consider the C_{geo} program below, which terminates on every iteration with probability $\frac{1}{2}$.

$$C_{\text{geo}} \triangleq \left\{ \begin{array}{l} x \coloneqq 0 \fatsemi i \coloneqq 0 \fatsemi \\ \textbf{while } x = 0 \textbf{ do} \\ \quad x :\approx \textbf{Ber}\left(\frac{1}{2}\right) \fatsemi \\ \quad i \coloneqq i + 1 \end{array} \right.$$

After n iterations, C_{geo} will have terminated with probability $1 - \frac{1}{2^n}$, which clearly tends towards 1, and the outcomes of the program form a *geometric* distribution, where $\mathbb{P}[i = n] = \frac{1}{2^n}$ for all $n > 0$. That distribution has an infinite support, so we cannot discover all the outcomes after a finite number of steps, but rather the full distribution is the supremum of all the finite approximations of the loop. Domain theory ensures that the supremum is always well defined.

The probabilistic powerdomain was first introduced by Jones and Plotkin [1989], whose construction was over continuous probability distributions. Here, we present a simpler discrete probabilistic powerdomain. Rather than working over $\mathcal{D}(X)$, as we did in the prior section, we now work over $\mathcal{D}(X_\bot)$ so as to also represent the probability of nontermination. First, we need a partial order to represent the notion of accumulating more probability mass for termination.

Lemma 2.11. *$\langle \mathcal{D}(X_\bot), \sqsubseteq_{\mathcal{D}} \rangle$ is a poset where:*

$$\mu \sqsubseteq_{\mathcal{D}} \nu \qquad \textit{iff} \qquad \forall x \in X.\ \mu(x) \leq \nu(x)$$

Proof. Follows immediately from the fact that $\langle \mathbb{R}, \leq \rangle$ is a poset. □

Note that in the definition of $\sqsubseteq_{\mathcal{D}}$, we quantify only over X and not $X_\bot$. It follows that $\mu \sqsubseteq_{\mathcal{D}} \nu$ implies that $\nu(\bot) \leq \mu(\bot)$, *i.e.,* as a distribution gets bigger, probability mass moves from $\bot$ to regular outcomes $x \in X$. Therefore, the bottom of the $\sqsubseteq_{\mathcal{D}}$ order is $\bot_{\mathcal{D}} \triangleq \delta_\bot$. Next, we also need a DCPO structure for $\mathcal{D}(X_\bot)$, which ensures that any directed set of distributions has a supremum.

Theorem 2.12. *$\langle \mathcal{D}(X_\perp), \sqsubseteq_{\mathcal{D}} \rangle$ is a DCPO where for any directed set $D \subseteq \mathcal{D}(X_\perp)$:*

$$(\sup D)(x) \triangleq \begin{cases} \sup_{\mu \in D} \mu(x) & \text{if } x \in X \\ \inf_{\mu \in D} \mu(\perp) & \text{if } x = \perp \end{cases}$$

Proof. Follows from Theorem 2.1 of Jones and Plotkin [1989]. □

In accordance with the definition of $\sqsubseteq_{\mathcal{D}}$, suprema maximize the probability of all $x \in X$ and minimize the probability of $\perp$. For example, after unrolling the loop in C_{geo} for n iterations starting in state σ, we get the following approximation of the final distribution.

$$\mu_n = \begin{bmatrix} \sigma[x \coloneqq 1, i \coloneqq 1] & \mapsto & \frac{1}{2} \\ \sigma[x \coloneqq 1, i \coloneqq 2] & \mapsto & \frac{1}{4} \\ \vdots & & \\ \sigma[x \coloneqq 1, i \coloneqq n] & \mapsto & \frac{1}{2^n} \\ \perp & \mapsto & \frac{1}{2^n} \end{bmatrix}$$

So $\mu_n(\tau) = \frac{1}{2^k}$ if $k \leq n$, $\tau(x) = 1$ (indicating that the loop terminated), $\tau(i) = k$, and otherwise τ is the same as σ. The condition that $k \leq n$ indicates that we only know the behavior of the first n iterations of the loop, so we do not yet know the probability that $i = k$ for any $k > n$. Since we do not yet have information about those outcomes, μ_n instead assigns probability $\frac{1}{2^n}$ to $\perp$, indicating that the loop has not yet terminated.

It is easy to see that $(\mu_n)_{n \in \mathbb{N}}$ is a chain, *i.e.,* that $\mu_n \sqsubseteq_{\mathcal{D}} \mu_{n+1}$ for all $n \in \mathbb{N}$, since each consecutive pair of distributions only differs in that μ_{n+1} assigns probability $\frac{1}{2^{n+1}}$ to the outcome where $i = n{+}1$, which is subtracted from the probability of $\perp$. Since any chain is a directed set, we can find $\sup_{n \in \mathbb{N}} \mu_n$ using the construction in Theorem 2.12 to obtain:

$$\sup_{n \in \mathbb{N}} \mu_n = \begin{bmatrix} \sigma[x \coloneqq 1, i \coloneqq 1] & \mapsto & \frac{1}{2} \\ \sigma[x \coloneqq 1, i \coloneqq 2] & \mapsto & \frac{1}{4} \\ \sigma[x \coloneqq 1, i \coloneqq 3] & \mapsto & \frac{1}{8} \\ \vdots & & \\ \sigma[x \coloneqq 1, i \coloneqq n] & \mapsto & \frac{1}{2^n} \\ \vdots & & \end{bmatrix}$$

In the supremum, the probability of $\bot$ goes to 0, since $\inf_{n \in \mathbb{N}} \frac{1}{2^n} = 0$. Instead, that probability mass is allocated over the infinite set of terminating outcomes.

The example above demonstrates an important property of probabilistic programs called *almost sure termination*, which means that the probability of termination converges to 1 in the limit. Formally, we can frame almost sure termination in terms of the program semantics as follows.

Definition 2.13 (Almost Sure Termination)**.** *The program C* almost surely terminates *in state σ if $[\![C]\!]_{\mathcal{D}}(\sigma)(\bot) = 0$. If C almost surely terminates in any start state state, then we say that it* universally *almost surely terminates.*

There are many approaches for proving almost sure termination [McIver and Morgan 2005; McIver et al. 2018; Majumdar and Sathiyanarayana 2025], which we will discuss in Section 4.3.3.

2.2.3 The Discrete Probability Monad

The next step is to define a monad instance for distributions, which gives semantics to sequential composition. The discrete probability monad is similar to the (continuous) Giry [1982] monad, it maps the continuation function $f : X \to \mathcal{D}(Y_\bot)$ over all the elements in the support of a distribution $\mu \in \mathcal{D}(X_\bot)$, and then combines the resulting distributions with a convex sum. Here, we present a variant of the discrete probability monad that is already composed with the nontermination monad.

Definition 2.14 (The Discrete Probability Monad)**.** $\left\langle \mathcal{D}((-)_\bot), \eta, (-)^\dagger \right\rangle$ *is a monad, where:*

$$\eta(x) \triangleq \delta_x \qquad\qquad f^\dagger(\mu) \triangleq \sum_{x \in \mathsf{supp}(\mu)} \mu(x) \cdot f_\bot(x)$$

and for any pointed poset $\langle Y, \leq \rangle$ with bottom $\bot_Y$ and function $f : X \to Y$, the bottom extension $f_\bot : X_\bot \to Y$ is defined as:

$$f_\bot(x) \triangleq \begin{cases} f(x) & \text{if } x \in X \\ \bot_Y & \text{if } x = \bot_X \end{cases}$$

To demonstrate why the weighted sum gives meaningful semantics, let us examine the resulting semantics of the program $x :\approx \mathbf{Ber}\left(\frac{2}{3}\right) \fatsemi y :\approx \mathbf{Ber}\left(\frac{x}{2}\right)$. Informally, we expect that x takes on the value 1 with probability $\frac{2}{3}$ and 0 with probability $\frac{1}{3}$. In the case that $x = 1$, y is distributed as a fair coin. So, by the definition of conditional probability, we get:

$$\mathbb{P}[x = 1 \wedge y = 1] \quad = \quad \mathbb{P}[x = 1] \cdot \mathbb{P}[y = 1 \mid x = 1] \quad = \quad \frac{2}{3} \cdot \frac{1}{2} \quad = \quad \frac{1}{3}$$

By the same logic, $\mathbb{P}[x = 1 \wedge y = 0] = \frac{1}{3}$ too. In the case that $x = 0$, which occurs with probability $1 - \frac{2}{3} = \frac{1}{3}$, y will be deterministically assigned to be 0, since $y = 1$ occurs with probability $\frac{x}{2} = \frac{0}{2} = 0$. So, in the end, we should expect that executing the program in state σ will result in a uniform distribution over the three states $\sigma[x := 1, y := 0]$, $\sigma[x := 1, y := 1]$, and $\sigma[x := 0, y := 0]$. The formal derivation of the semantics below exactly matches the intuition.

$$\begin{aligned}
&\left[\!\left[x :\approx \mathbf{Ber}\left(\tfrac{2}{3}\right) \fatsemi y :\approx \mathbf{Ber}\left(\tfrac{x}{2}\right)\right]\!\right](\sigma) \\
&= \left[\!\left[y :\approx \mathbf{Ber}\left(\tfrac{x}{2}\right)\right]\!\right]^{\dagger}\left(\left[\!\left[x :\approx \mathbf{Ber}\left(\tfrac{2}{3}\right)\right]\!\right](\sigma)\right) \\
&= \left[\!\left[y :\approx \mathbf{Ber}\left(\tfrac{x}{2}\right)\right]\!\right]^{\dagger}\left(\begin{bmatrix} \sigma[x := 1] & \mapsto & 2/3 \\ \sigma[x := 0] & \mapsto & 1/3 \end{bmatrix}\right) \\
&= \frac{2}{3}\left[\!\left[y :\approx \mathbf{Ber}\left(\tfrac{x}{2}\right)\right]\!\right](\sigma[x := 1]) + \frac{1}{3}\left[\!\left[y :\approx \mathbf{Ber}\left(\tfrac{x}{2}\right)\right]\!\right](\sigma[x := 0]) \\
&= \frac{2}{3}\left(\begin{bmatrix} \sigma[x := 1, y := 1] & \mapsto & 1/2 \\ \sigma[x := 1, y := 0] & \mapsto & 1/2 \end{bmatrix}\right) + \frac{1}{3}\left(\begin{bmatrix} \sigma[x := 0, y := 1] & \mapsto & 0 \\ \sigma[x := 0, y := 0] & \mapsto & 1 \end{bmatrix}\right) \\
&= \begin{bmatrix} \sigma[x := 1, y := 1] & \mapsto & 1/3 \\ \sigma[x := 1, y := 0] & \mapsto & 1/3 \\ \sigma[x := 0, y := 0] & \mapsto & 1/3 \end{bmatrix}
\end{aligned}$$

By Theorem C.3 of Li et al. [2025b], the Kleisli extension from Definition 2.14 is Scott continuous, meaning that it is suitable for us to use as a basis for a denotational model.

2.3 HOARE LOGIC

Hoare Logic is a deductive reasoning system for specifying programs in terms of preconditions and postconditions [Hoare 1969]. Roughly speaking, a Hoare Triple $\{P\}\ C\ \{Q\}$ means that if the program C is run in an initial state satisfying P, then any reachable end state will satisfy the postcondition Q. We formalize the validity of Hoare Triples using the powerdomain semantics

$$\frac{}{\{P\}\ \texttt{skip}\ \{P\}}\textsc{Skip} \qquad \frac{}{\{P[e/x]\}\ x \coloneqq e\ \{P\}}\textsc{Assign} \qquad \frac{\{P\}\ C_1\ \{Q\} \qquad \{Q\}\ C_2\ \{R\}}{\{P\}\ C_1 \fatsemi C_2\ \{R\}}\textsc{Seq}$$

$$\frac{\{P \wedge b\}\ C_1\ \{Q\} \qquad \{P \wedge \neg b\}\ C_2\ \{Q\}}{\{P\}\ \texttt{if}\ b\ \texttt{then}\ C_1\ \texttt{else}\ C_2\ \{Q\}}\textsc{If} \qquad \frac{\{P \wedge b\}\ C\ \{P\}}{\{P\}\ \texttt{while}\ b\ \texttt{do}\ C\ \{P \wedge \neg b\}}\textsc{While}$$

$$\frac{P' \Rightarrow P \quad \{P\}\ C\ \{Q\} \quad Q \Rightarrow Q'}{\{P'\}\ C\ \{Q'\}}\textsc{Consequence}$$

Figure 2.5: The inference rules of Hoare Logic, where $P[e/x]$ denotes syntactic substitution of the expression e for the variable x in the assertion P.

$[\![C]\!] : \mathsf{Mem}[V] \to 2^{\mathsf{Mem}[V]}$ from Section 2.1. Suppose that P and Q belong to some assertion syntax (we will introduce a concrete one in Section 2.4) and $\sigma \vDash P$ means that $\sigma \in \mathsf{Mem}[V]$ satisfies the assertion P. Then a Hoare Triple is valid, written $\vDash \{P\}\ C\ \{Q\}$ iff:

$$\forall \sigma \vDash P. \quad \forall \tau \in [\![C]\!]\,(\sigma). \quad \tau \vDash Q$$

Hoare Logic captures *partial correctness*, Q will hold at the end of *any* terminating execution, but Hoare Logic does not guarantee termination. If termination is important, then Total Hoare Logic [Manna and Pnueli 1974] can instead be used, which has a slightly different metatheory.

Inference rules to deductively derive Hoare Triples are shown in Figure 2.5. Many of the rules are straightforward, for example, the Skip rule states that a no-op preserves the precondition, and Seq allows one to derive a specification for a sequential composition from specifications of each command in the sequence.

The While rule internalizes the partial correctness nature of Hoare Logic. To specify the behavior of a loop, one only needs to find an *invariant* P, which is preserved by each iteration of the loop body. If such an invariant exists, then it will remain true after termination, although termination is not guaranteed. In order to establish termination, more sophisticated rules are needed, which incorporate a strictly decreasing *rank*, which witnesses that each iteration of the loop moves it closer to termination.

Finally, the rule of Consequence allows for strengthening of the precondition and weakening of the postcondition, which is useful in order to abstract away from the concrete program state and express specifications in a simpler and more intuitive form.

Cook [1978] showed that Hoare Logic is *relatively complete*, meaning that the inference rules in Figure 2.5 are sufficient to prove any valid specification, modulo an oracle that can decide implications in the rule of Consequence.[1] But despite Cook's result, Hoare Logic does not provide the *compositional* reasoning principles needed to make program analysis scale. In the proceeding sections, we will see how modern extensions to Hoare Logic enable large scale reasoning.

2.4 SEPARATION LOGIC

Separation logic is an extension of Hoare Logic, which provides compositional mechanisms to reason about mutable state [Reynolds 2002; O'Hearn et al. 2001]. That compositionality has brought great success in multiple domains including large scale static analysis [Calcagno et al. 2009, 2011, 2015; Distefano et al. 2019] and deductive reasoning about concurrency [O'Hearn 2004; Brookes 2004; Jung et al. 2018; Carbonneaux et al. 2022].

Separation logic's key innovation is the *separating conjunction* $P * Q$, a new logical connective, which guarantees that P and Q are satisfied in disjoint portions of memory. Separation allows for local reasoning wherein properties about program fragments only concern their local resources. Locality gives rise to compositionality, since specifications can be lifted into a larger memory footprint with the guarantee that no resources outside of the local footprint will be affected by execution of the program. In the remainder of this section, we will formalize a fragment of separation logic and see how it enables scalable reasoning.

[1] As shown by Lipton [1977], implications are undecidable in any sufficiently *expressive* assertion language for Hoare Logic, leading to Cook's [1978] notion of *relative* completeness. For a complete discussion, refer to Apt [1981].

2.4.1 Syntax and Semantics of Separation Logic Assertions

Although separation logic was originally formulated to reason about heaps, in this thesis we will work in a simplified model based on variable stores. The reason for the simplified approach is that it is not yet well understood how heaps and dynamically allocated memory interact with the combination of probabilistic computation, concurrency, and unbounded loops.

Our separation logic model is built around memories (Definition 2.1). The disjoint union $\uplus\colon \mathsf{Mem}[U] \to \mathsf{Mem}[V] \to \mathsf{Mem}[U \cup V]$ combines two memories as long as $U \cap V = \emptyset$, and a similar operation $A * B$ is defined on sets of memories $A \subseteq \mathsf{Mem}[U]$ and $B \subseteq \mathsf{Mem}[V]$.

$$(\sigma \uplus \tau)(x) \triangleq \begin{cases} \sigma(x) & \text{if } x \in U \\ \tau(x) & \text{if } x \in V \end{cases} \qquad\qquad A * B \triangleq \{\sigma \uplus \tau \mid \sigma \in A, \tau \in B\}$$

We define projections $\pi_U\colon \mathsf{Mem}[V] \to \mathsf{Mem}[U \cap V]$ as $\pi_U(\sigma)(x) \triangleq \sigma(x)$ if $x \in U$. The syntax for assertions is shown below.

$$P ::= \mathbf{true} \mid \mathbf{false} \mid P \wedge Q \mid P \vee Q \mid P * Q \mid \exists X.\, P \mid e \mapsto E \mid E_1 \asymp E_2 \qquad (\asymp \in \{=, \leq, <, \in, \ldots\})$$
$$E ::= X \mid v \mid E_1 + E_2 \mid E_1 \cdot E_2 \mid \cdots$$

In addition to expressions and variables from Section 6.2, assertions also depend on logical variables $X, Y, Z \in \mathsf{LVar}$, which cannot be modified by programs. Logical expressions $E \in \mathit{lexp}$ mirror standard ones, but operate over logical variables $X \in \mathsf{LVar}$ rather than $x \in \mathsf{Var}$. Logical expression evaluation $[\![E]\!]_{\mathit{lexp}}(\Gamma)$ in a context $\Gamma\colon \mathsf{LVar} \to \mathsf{Val}$ is defined in a standard way.

Separation logic assertions are modeled by both a context Γ, and a memory $\sigma \in \mathsf{Mem}[V]$, the satisfaction relation is shown in Figure 2.6. The meaning of **true**, **false**, conjunction, and disjunction are standard. The separating conjunction $P * Q$ means that the memory $\sigma \in \mathsf{Mem}[V]$ can be divided into two smaller memories $\sigma_1 \in \mathsf{Mem}[V_1]$ and $\sigma_2 \in \mathsf{Mem}[V_2]$ to satisfy P and Q individually. By the definition of $\uplus$, V_1 and V_2 must be disjoint. Our logic is an *intuitionistic* [Docherty 2019] or *affine* interpretation of separation logic, meaning that information about variables can be discarded; if $\Gamma, \sigma \vDash P$, then P need not describe the entire memory σ. As such,

$$
\begin{array}{lll}
\Gamma, \sigma \vDash \textbf{true} & \text{always} \\
\Gamma, \sigma \vDash \textbf{false} & \text{never} \\
\Gamma, \sigma \vDash P \wedge Q & \text{iff} \quad \Gamma, \sigma \vDash P \quad \text{and} \quad \Gamma, \sigma \vDash Q \\
\Gamma, \sigma \vDash P \vee Q & \text{iff} \quad \Gamma, \sigma \vDash P \quad \text{or} \quad \Gamma, \sigma \vDash Q \\
\Gamma, \sigma \vDash P * Q & \text{iff} \quad \exists \sigma_1, \sigma_2.\ \ \sigma_1 \uplus \sigma_2 \sqsubseteq \sigma \quad \text{and} \quad \Gamma, \sigma_1 \vDash P \quad \text{and} \quad \Gamma, \sigma_2 \vDash Q \\
\Gamma, \sigma \vDash \exists X.\, P & \text{iff} \quad \Gamma[X := v], \sigma \vDash P \text{ for some } v \in \mathsf{Val} \\
\Gamma, \sigma \vDash e \mapsto E & \text{iff} \quad [\![e]\!]_{exp}\,(\sigma) = [\![E]\!]_{\mathsf{LExp}}\,(\Gamma) \\
\Gamma, \sigma \vDash E_1 \asymp E_2 & \text{iff} \quad [\![E_1]\!]\,(\Gamma) \asymp [\![E_2]\!]\,(\Gamma)
\end{array}
$$

Figure 2.6: Satisfaction relation for separation logic assertions.

we only require that $\sigma_1 \uplus \sigma_2 \sqsubseteq \sigma$, which we define as $\sigma \sqsubseteq \tau$ iff $\sigma \uplus \sigma' = \tau$ for some σ', so τ could contain more variables than σ.

We also include a points-to predicate $e \mapsto E$, although it has a slightly different meaning than points-to predicates in the standard heap model of separation logic [Reynolds 2002]. Here, e does not describe a pointer, but can rather be any concrete expression, and $e \mapsto E$ simply means that the e evaluates to the same value under σ as E does under Γ, allowing us to connect concrete and logical state. Finally, $E_1 \asymp E_2$ allows us to make assertions about logical state, where $\asymp \in \{=, \leq, \in, \cdots\}$ ranges over similar comparators to the ones in the syntax of assertions P. We define the notation $(\!|P|\!)^V_\Gamma$ to obtain the set of all memories $\sigma \in \mathsf{Mem}[V]$ that satisfy an assertion P, and we omit the superscript when we wish to minimize V, so that the memories contain only the free variables of P:

$$(\!|P|\!)^V_\Gamma \triangleq \{\sigma \in \mathsf{Mem}[V] \mid \Gamma, \sigma \vDash P\} \qquad\qquad (\!|P|\!)_\Gamma \triangleq (\!|P|\!)^{\mathsf{vars}(P)}_\Gamma$$

Finally, we provide syntactic sugar for restricting the domain of existential quantifiers, asserting membership in a set, and asserting that the resources of e are owned by the current thread.

$$\exists X \in E.\, P \triangleq \exists X.\, P * X \in E \qquad e \in E \triangleq \exists X \in E.\, e \mapsto X \qquad e \mapsto - \triangleq \exists X. e \mapsto X$$

$$\mathsf{own}(e_1, \ldots, e_n) \triangleq e_1 \mapsto - * \cdots * e_n \mapsto -$$

2.4.2 Local Reasoning and the Frame Rule

Separation logic's compositionality is built around the idea of *local reasoning*, the ability to prove properties about program fragments in their local footprints, and then lift those proofs into a global context. As such, separation logic provides *small axioms* for reasoning about actions, such as the rule for assignment below [O'Hearn et al. 2001].

$$\frac{}{\{x \mapsto -\}\ x := v\ \{x \mapsto v\}}\textsc{Assign}$$

Unlike the Assign rule of Hoare Logic (Figure 2.5), which operates in a global context, the new rule above only reasons *locally* about the variable x. The precondition of separation logic's Assign rule asserts *ownership* of the variable x, ensuring that no one else can retain any information about it. Therefore, the following triple is also valid for any *frame* F.

$$\{x \mapsto - * F\}\ x := v\ \{x \mapsto v * F\}$$

Since x is separated from F, we know that the modification of x cannot affect the truth of F. In fact, the reasoning above is valid not only for individual variable assignments, but also for any arbitrary command, giving rise to the Frame rule.

$$\frac{\{P\}\ C\ \{Q\}}{\{P * F\}\ C\ \{Q * F\}}\textsc{Frame}$$

The Frame rule has formed the basis for scalable static analysis techniques by providing a new dimension of compositionality of top of Hoare Logic. Hoare Logic already supports sequential compositionality via the Seq rule. However, Seq requires that the postcondition of the first command exactly matches the precondition of the next command. In a large scale static analysis, the commands may be complex procedure calls, which already have precomputed specifications that do not match each other.

The spatial compositionality afforded by the frame rule provides a strategy for stringing together sequences of arbitrary specifications. That composition is achieved with the help of *bi-abduction*, a heuristic decision procedure for inferring the leftover frame F and missing anti-frame

M to make the following entailment hold [Calcagno et al. 2009, 2011].

$$P * [M] \vdash Q * [F]$$

Using bi-abduction, the following derived rule can infer a specification for a sequentially composed program $C_1 \fatsemi C_2$ even if the postcondition of C_2 does not match the precondition of C_1.

$$\frac{\{P_1\}\ C_1\ \{Q_1\} \qquad Q_1 * [M] \vdash P_2 * [F] \qquad \{P_2\}\ C_2\ \{Q_2\}}{\{P_1 * M\}\ C_1 \fatsemi C_2\ \{Q_2 * F\}}\textsc{Bi-Abductive Seq}$$

Meta's Infer Static Analyzer is built around bi-abduction [Calcagno et al. 2015], which has allowed it to scale to efficiently analyze massive codebases [Distefano et al. 2019]. Infer routinely find bugs in code changes at Meta for software written in a variety of languages including C, C++, Java, and Hack. Although we do not explore static analysis in this thesis, the compositionality benefits of the Frame rule still carry over to probabilistic contexts, as first explored by Barthe et al. [2020].

2.4.3 Concurrent Separation Logic

Separation logic has further applications to verification of concurrent programs. One key difficulty in reasoning concurrency is that the semantics of programs is not compositional; the behavior of two threads can completely change when they are composed in parallel. Take, for example, the following concurrent program.

$$\begin{array}{l} x \coloneqq 0 \fatsemi \\ \textbf{if}\ x = 0\ \textbf{then} \\ \quad y \coloneqq 0 \\ \textbf{else} \\ \quad y \coloneqq 1 \end{array} \quad \Big\| \quad x \coloneqq 1$$

Executing the left thread on its own, we would expect that x and y are always equal to 0. Meanwhile, executing the right thread on its own would yield $x = 1$, and y left untouched. But executing both threads in parallel unlocks a new branch of control flow. If $x \coloneqq 1$ is scheduled between $x \coloneqq 0$ and the if statement, then y will be set to 1, a behavior that was not observable by executing either thread on its own.

Despite the non-compositional semantics, compositional reasoning about concurrency is possible with separation logic. The key idea is to use separation to guarantee that each thread operates in its own region of memory, so that the threads are guaranteed not to interfere with each other. Inferences about the behavior of a parallel program can therefore be made from specifications about each thread without worrying about all the possible ways that they can be interleaved. All of that intuition is encapsulated by the PAR rule of Concurrent Separation Logic (CSL) [O'Hearn 2004; Brookes 2004].

$$\frac{\{P_1\}\ C_1\ \{Q_1\} \qquad \{P_2\}\ C_2\ \{Q_2\}}{\{P_1 * P_2\}\ C_1 \parallel C_2\ \{Q_1 * Q_2\}}\textsc{Par}$$

Of course, additional difficulties in concurrency analysis come with shared state, when the memory footprints of the two threads cannot be fully separated. To deal with shared state, CSL introduces the concept of resource invariants, which are assertions about the shared state that must remain true after every atomic step of computation.

More recently, Iris has introduced a generalized notion of resources in order to express more sophisticated logical properties about shared state [Jung et al. 2018, 2015]. For example, a particular memory configuration may correspond to logical ownership of a lock, so that one particular thread is allowed to freely modify certain memory locations. In Chapter 7, we will see how the notion of resource invariants carries over into the probabilistic setting.

2.5 OUTCOME LOGIC

Outcome Logic is a generalized foundation for unified reasoning about programs whose execution branches due to different kinds of computational effects [Zilberstein 2025]. The unification is based on *weighted programming*, a paradigm that uses different kinds of weights to encode different kinds of computation [Batz et al. 2022]. Programs are interpreted in a semantic domain that maps each state (in our case, memories) to a weight of type W.

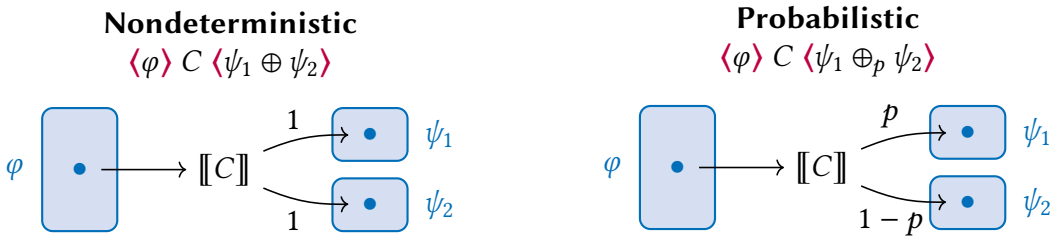
$$[\![C]\!] : \mathsf{Mem}[V] \to (\mathsf{Mem}[V] \to W)$$

As long as W is a *partial semiring* $\langle W, +, \cdot, \mathbb{1}, \mathbb{0} \rangle$, meaning that it has addition and multiplication operations with associated identities, then the program semantics is defined in exactly the same way as we saw in Section 2.1, where the monad instance is a generalized version of the probabilistic monad (Definition 2.14). Roughly speaking, addition corresponds to choice between two branches of computation, and multiplication corresponds to accumulating weight for an outcome in a sequence of commands.

Nondeterministic computation is captured by the Boolean semiring $\langle \mathbb{B}, \vee, \wedge, 0, 1 \rangle$, which uses Boolean weights to express whether or not each memory is a possible outcome of running the program. Addition is given by logical disjunction, meaning that a memory is a possible outcome if it is possible in any of the computation branches. Multiplication is given by conjunction because a memory τ is only reachable via a sequential composition $C_1 \fatsemi C_2$ if there is some σ which is reachable by running C_1 *and* τ is also reachable via $[\![C_2]\!](\sigma)$. Indeed, the set of functions of type $\mathsf{Mem}[V] \to \mathbb{B}$ is isomorphic to the powerset $2^{\mathsf{Mem}[V]}$, so the weighted programming approach is equivalent to Example 2. Probabilistic computation is captured by the Probabilistic semiring $\langle [0, 1], +, \cdot, 0, 1 \rangle$, which results exactly in the discrete probability monad (Definition 2.14).

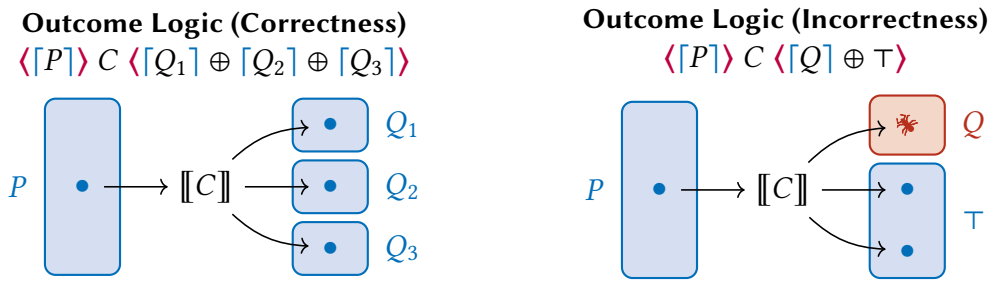
Outcome Logic is similar to Hoare Logic in that it specifies programs in terms of precondition and postconditions. However, unlike Hoare Logic, outcome triples $\langle \varphi \rangle\ C\ \langle \psi \rangle$ expose the program weights in the assertion language, giving a far greater degree of expressivity. That expressivity is enabled a new logical connective, the *outcome conjunction* $\varphi \oplus \psi$, which states that φ and ψ are two distinct outcomes that can result from running the program. The outcome conjunction works in tandems with the weights to express different kinds of properties. For example, we can express that two outcomes are possible by running a nondeterministic program, and we can

precisely quantify the likelihood of two outcomes after running a probabilistic program.



Outcome Logic was initially developed to unify the theories of correctness and incorrectness [Zilberstein et al. 2023], which had previously diverged into the separate and incompatible Hoare Logic and Incorrectness Logic [O'Hearn 2020]. The key difference is that a program is correct only if all executions are safe, whereas it is incorrect even if just one execution leads to a bug. Over the last several years, the study of theories of incorrectness has ramped up, in part to ensure that bug finding analyses such as Infer produce only true positives [Raad et al. 2020; Le et al. 2022].

In Outcome Logic, it is quite easy to express both correctness properties (which apply to all outcomes) and also incorrectness properties (which single out an erroneous outcome).



The Outcome Logic approach to incorrectness has additional benefits, such as making it easier to identify the context that will trigger a bug to occur [Zilberstein et al. 2023]. In addition, Outcome Logic is compatible with bi-abduction [Zilberstein et al. 2024], the technique that has allowed Infer to scale to the analysis of massive codebases.

Outcome Logic's weighted programming approach is quite extensible in that the underlying model can be composed with additional effects such as exceptions [Zilberstein et al. 2023], mutable state [Zilberstein et al. 2024], nontermination [Li et al. 2025b; Li 2025], and multiple types of outcomes in tandem such as nondeterminism and randomization [Zilberstein et al. 2025b]. In the

remainder of this thesis, we will build on the probabilistic instance of Outcome Logic. Starting in the next chapter, we will see how Outcome Logic together with separation logic provides an expressive and compositional foundation for reasoning about probabilistic programs. Throughout the remaining chapters, we will see how nondeterminism and concurrency are incorporated into that initial formulation.

CHAPTER 3

A MEASURE THEORETIC MODEL OF OUTCOMES

As we saw in the previous chapter, Hoare [1969] style program logics specify programs in terms of preconditions and postconditions. In more expressive logics like separation logic and Outcome Logic, the pre- and postconditions can express not only the values of variables, but also additional information such as disjointness of memory locations and the probabilities of different possible outcomes. In this chapter, we formally define the assertion language that will be used in the logics that we introduce throughout the remaining chapters.

3.1 OVERVIEW

Unlike traditional programs, probabilistic programs produce a *distribution* of outcomes. Thus, formal methods for probabilistic programs must provide a language for describing each outcome and its likelihood. A few classes of programs that are most naturally described in terms of outcomes include:

- ***Encryption Algorithms***, where keys are uniformly sampled and the resulting ciphertext must accordingly be uniformly distributed.
- ***Randomness Conserving Protocols***, where a high-quality source of randomness is derived from low-quality sources of randomess [von Neumann 1951; Kozen and Soloviev 2018].
- ***Shuffling Algorithms***, where the input data must be uniformly permuted at the end of execution [Fisher and Yates 1938; Bacher et al. 2018].

In addition to describing outcomes, compositionality plays an important role in deductive program analysis. Rather than describing the entire space of outcomes in a single irreducible assertion, we need the ability to describe parts of the distribution, and to combine those parts into a full view. To that end, *probabilistic independence* proves crucial, as it allows us to reconstruct the *joint distribution* over all the program variables without explicitly stating their relative probabilities.

In this chapter, we introduce an assertion language for reasoning about probabilistic programs in terms of outcomes and independence. First, we discuss the overarching ideas behind that assertion language, and then in later sections we formalize the semantics using measure theory.

3.1.1 Specifying Distributions via Outcomes

Outcome assertions provide an expressive language for describing the results of probabilistic computations. The key primitive for describing a single outcome is the pure assertion $\lceil P \rceil$, which lifts standard separation logic assertions P from Section 2.4 to be an outcome assertion, which holds with probability 1. For example, after running the program $x \coloneqq 5$, we can state that $\lceil x \mapsto 5 \rceil$, *i.e.,* x has the value 5 with probability 1.

The *outcome conjunction* $\bigoplus_{X \sim d} \varphi$ expresses that the program has several possible behaviors, each occurring with a precise probability. It binds a new logical variable X, which is distributed according to the distribution d, and φ holds for each value in the support of d. For example, the following assertion states that x is distributed according to a Bernoulli distribution with parameter $\frac{1}{2}$ (a fair coin flip).

$$\bigoplus_{X \sim \mathbf{Ber}\left(\frac{1}{2}\right)} \lceil x \mapsto X \rceil$$

The outcome conjunction presented here is an evolution of the one that we saw in Section 2.5 and is now specialized to the probabilistic setting. It also draws inspiration from the *conditioning modalities* of Lilac [Li et al. 2023] and Bluebell [Bao et al. 2025], although the semantic foundations are set up differently to enable compositional reasoning about outcomes. See Section 3.4 for a more complete comparison. As an abbreviation, we use $x \sim d$ as a shorthand for $\bigoplus_{X \sim d} \lceil x \mapsto X \rceil$.

3.1.2 Independence as Compositionality

As we have already seen, compositionality is the key to scalable program analysis. In the case of stateful and concurrent programs, separation and *spatial* compositionality allow us to reason modularly about components of a program, and then extrapolate that behavior into a larger mem-

ory footprint. Moving to the probabilistic setting, a second dimension of modularity is needed. We not only need to know about the value of each variable, but also how those values are *distributed*. And we not only need to know how they are distributed with respect to the local memory, but also the *joint distribution* over all the variables in the entire global memory.

To demonstrate, consider the sequential composition of two simple programs, each of which flips a fair coin and stores the result in the variables x and y, respectively.

$$x :\approx \mathbf{Ber}\left(\tfrac{1}{2}\right) \qquad \fatsemi \qquad y :\approx \mathbf{Ber}\left(\tfrac{1}{2}\right)$$

As a result, we may wish to say that x is distributed according to $\mathbf{Ber}\left(\frac{1}{2}\right)$ and, in a separate portion of memory, y is also distributed according to $\mathbf{Ber}\left(\frac{1}{2}\right)$.

$$\left(x \sim \mathbf{Ber}\left(\tfrac{1}{2}\right)\right) * \left(y \sim \mathbf{Ber}\left(\tfrac{1}{2}\right)\right)$$

As we previously saw in Section 2.4, the separating conjunction $*$ guarantees that x and y are stored in separate portions of memory, so that mutations to one will not affect the other. The individual probabilistic assertions also tell us how x and y are individually distributed. But we are still missing a crucial detail: *what is the joint distribution over x and y?*

To answer that question, we work in a model first explored by Barthe et al. [2020] where separation also models probabilistic independence. In classical probability theory, independence is a property of *events* or sets of outcomes in a random experiment. For now, we can think of an event as being a separation logic assertion P, which indeed describes a set of memories. Two events P and Q are independent if the probability of both events occurring is equal to the product of the individual probabilities of each event.

$$\mathbb{P}[P * Q] = \mathbb{P}[P] \cdot \mathbb{P}[Q]$$

In the case of program above, each sampling operation is probabilistically independent, so for example, the probability that $x \mapsto 1$ and $y \mapsto 1$ can be obtained by multiplication:

$$\mathbb{P}[x \mapsto 1 * y \mapsto 1] \quad = \quad \mathbb{P}[x \mapsto 1] \cdot \mathbb{P}[y \mapsto 1] \quad = \quad \frac{1}{2} \cdot \frac{1}{2} \quad = \quad \frac{1}{4}$$

Multiplication of the probabilities of each event is precisely what is achieved by the probabilistic separating conjunction; the assertion $\left(x \sim \mathbf{Ber}\left(\frac{1}{2}\right)\right) * \left(y \sim \mathbf{Ber}\left(\frac{1}{2}\right)\right)$ means that the joint distribution over x and y can be obtained by multiplying the probabilities of the individual outcomes. In fact, the independence property means that the following distributive rule holds; as long as X is not free in ψ, we can move ψ inside of an outcome conjunction in the same way as multiplication distributes over addition in standard arithmetic.

$$\left(\bigoplus_{X \sim d} \varphi\right) * \psi \qquad \Longrightarrow \qquad \bigoplus_{X \sim d} (\varphi * \psi)$$

In the case of the prior example, that we can alternatively express the postcondition as an iterated outcome conjunction.

$$\left(x \sim \mathbf{Ber}\left(\frac{1}{2}\right)\right) * \left(y \sim \mathbf{Ber}\left(\frac{1}{2}\right)\right) \Longrightarrow \bigoplus_{X \sim \mathbf{Ber}\left(\frac{1}{2}\right)} \bigoplus_{Y \sim \mathbf{Ber}\left(\frac{1}{2}\right)} \lceil x \mapsto X * y \mapsto Y \rceil$$

The independence-as-compositionality property will repeatedly play an important role throughout the examples in this thesis, for example, to ensure proper distributions of outcomes in randomness-conserving protocols, and modular shuffling algorithms.

3.1.3 Granularity of Independence and Measurable Events

Given that independence is a property of events, questions remain over how granular the independence must be in order to allow separation. If two variables x and y are correlated, is there any valid way to express a specification about x and y with a separating conjunction?

The answer to that question depends on how granularly we need to specify the distributions over x and y, and also the nature of the correlation between them. For example, suppose x and y are generated via the following program.

$$x \coloneqq 1 \qquad \fatsemi \qquad y \coloneqq x$$

Clearly x and y are correlated, since y is directly derived from x. But x and y are also deterministic, making the independence property trivial. Since the probability of any event is either 0 or 1, all

of the arithmetic needed to ensure independence works out properly. For the probability 1 event, the individual probabilities multiply to the desired value.

$$\mathbb{P}[x \mapsto 1 * y \mapsto 1] \quad = \quad 1 \quad = \quad 1 \cdot 1 \quad = \quad \mathbb{P}[x \mapsto 1] \cdot \mathbb{P}[y \mapsto 1]$$

And the same is true of a probability 0 event, for example:

$$\mathbb{P}[x \mapsto 1 * y \mapsto 0] \quad = \quad 0 \quad = \quad 1 \cdot 0 \quad = \quad \mathbb{P}[x \mapsto 1] \cdot \mathbb{P}[y \mapsto 0]$$

Stemming from the above example, it is completely equivalent to put a separating conjunction inside or outside of the pure assertion brackets $\lceil - \rceil$.

$$\lceil P * Q \rceil \qquad \Longleftrightarrow \qquad \lceil P \rceil * \lceil Q \rceil$$

However, the situation is a bit more complicated when the variables display nontrivial correlation, such as in the program below.

$$x :\approx \mathbf{Ber}\left(\tfrac{1}{2}\right) \qquad ; \qquad y \coloneqq x$$

Here, we have a clear counterexample to the independence property.

$$\mathbb{P}[x \mapsto 0 * y \mapsto 0] \quad = \quad \frac{1}{2} \quad \neq \quad \frac{1}{2} \cdot \frac{1}{2} \quad = \quad \mathbb{P}[x \mapsto 0] \cdot \mathbb{P}[y \mapsto 0]$$

Therefore, $\left(x \sim \mathbf{Ber}\left(\tfrac{1}{2}\right)\right) * \left(y \sim \mathbf{Ber}\left(\tfrac{1}{2}\right)\right)$ is not a valid postcondition for the program above. Of course, as we saw in Section 3.1.1, we can specify the program above with the outcome conjunction $\bigoplus_{X \sim \mathbf{Ber}\left(\frac{1}{2}\right)} \lceil x \mapsto X * y \mapsto X \rceil$, but sometimes we will also need to completely separate the information about x and y, for example, if one of those variables is part of the shared state in a concurrent program.

We can achieve separation here by decreasing the precision in the probabilistic events that we choose to measure. For instance, if we do not wish to assert exactly how y is distributed (*i.e.*, $y \sim \mathbf{Ber}\left(\tfrac{1}{2}\right)$), but instead only assert a probability 1 event about all the possible outcomes $\lceil y \in \{0, 1\} \rceil$, then the independence property still holds.

$$\mathbb{P}[x \mapsto 0 * y \in \{0, 1\}] \quad = \quad \frac{1}{2} \quad = \quad \frac{1}{2} \cdot 1 \quad = \quad \mathbb{P}[x \mapsto 0] \cdot \mathbb{P}[y \in \{0, 1\}]$$

Therefore, we can specify the program with the assertion $(x \sim \mathbf{Ber}\left(\frac{1}{2}\right)) * \lceil y \in \{0,1\} \rceil$. Although less precise, the information about x and y is now fully separated, opening up possibilities for modular reasoning. In practice, the type of assertion about the distribution of states will often depend highly on the domain, and our measure theoretic model of outcome assertions accordingly supports a variety of compositional reasoning principles.

3.2 PRELIMINARIES

We now introduce concepts from measure theory, which will be needed to formalize our model of outcomes. For a more in depth introduction, refer to Royden [1968] or Fremlin [2001].

3.2.1 Probability Spaces

Probability spaces are mathematical representations of a random process, which precisely define not only the probabilities of events but also which events are measurable. Probability spaces are typically used to define continuous distributions, such as the uniform distribution over the unit interval $[0,1]$, where each sample (*i.e.,* point) has probability 0, but nonempty intervals have nonzero probabilities. For example, a sample from the uniform distribution over $[0,1]$ has probability $\frac{1}{2}$ of lying in the interval $[\frac{1}{4}, \frac{3}{4}]$.

In this thesis, we are concerned only with analyzing *discrete* randomized programs, where each of the countably many outcomes has a distinct nonzero probability. However, probability spaces are also useful in the discrete case for describing how much information is available about a given distribution, which allows us to express varying levels of probabilistic independence as we saw in Section 3.1.3.

Each probability space ranges over a universe Ω, which is the set of all *samples*. In the case of the above example, the universe is the unit interval $[0,1]$. However, there may not be enough information to determine the probability of any individual sample, it may instead only be possible to measure the probability of some *events*, which are sets of samples. So, the probability space

must also contain a set of events $\mathcal{F}$. The event space must be a σ-algebra, which means that it contains $\emptyset$ and Ω, and is closed under complementation and countable unions.

Definition 3.1 (σ-Algebra)**.** *Given a universe of samples Ω, a σ-algebra $\mathcal{F} \subseteq 2^{\Omega}$ on Ω is a set of subsets of Ω such that:*

1. $\Omega \in \mathcal{F}$
2. *$\mathcal{F}$ is closed under complementation: if $A \in \mathcal{F}$, then $\Omega \setminus A \in \mathcal{F}$*
3. *$\mathcal{F}$ is closed under countable unions: if $A_i \in \mathcal{F}$ for all $i \in I$ (where I is a countable index set), then $\bigcup_{i \in I} A_i \in \mathcal{F}$.*

The events $\emptyset$ and Ω are always measurable, as they have probabilities 0 and 1 respectively. In addition, the complement of an event A is measurable too: if A has probability p, then $\Omega \setminus A$ has probability $1-p$. Countable unions of measurable events must be measurable, and by de Morgan's law, that implies that countable intersections are also measurable. The aforementioned probabilities are given by a *probability measure* $\mu \colon \mathcal{F} \to [0, 1]$, which is a mapping from measurable events to probabilities.

Definition 3.2 (Probability Measure)**.** *Given a σ-algebra $\mathcal{F}$ on the sample space Ω, a probability measure is a function $\mu \colon \mathcal{F} \to \mathbb{R}$ from $\mathcal{F}$ to the real numbers $\mathbb{R}$ such that:*

1. $\mu(\emptyset) = 0$
2. $\mu(\Omega) = 1$
3. *μ obeys countable additivity: for any countable sequence $(A_i)_{i \in I}$ of pairwise disjoint sets, $\mu(\biguplus_{i \in I} A_i) = \sum_{i \in I} \mu(A_i)$*

Countable additivity formalizes the fact that the probability of a sample coming from one of a disjoint collection of events can be obtained by adding the individual probabilities. For example, a uniform sample from the unit interval is in $[0, \frac{1}{3}] \cup [\frac{2}{3}, 1]$ with probability $\frac{1}{3} + \frac{1}{3} = \frac{2}{3}$. A probability space is simply a triple containing a sample space, an event space, and a probability measure.

Definition 3.3 (Probability Spaces)**.** *A probability space* $\mathcal{P} = \langle \Omega, \mathcal{F}, \mu \rangle$ *is a 3-tuple containing a sample space* Ω*, an event space* $\mathcal{F} \in 2^{2^\Omega}$*, and a probability measure* $\mu \colon \mathcal{F} \to [0, 1]$

Going forward, we refer to the components of a probability space $\mathcal{P}$ as $\Omega_\mathcal{P}$, $\mathcal{F}_\mathcal{P}$, and $\mu_\mathcal{P}$. We require probability spaces to be *complete*, meaning that they contain all events of measure zero. More formally, $\mathcal{P}$ is complete if for any $A \in \mathcal{F}_\mathcal{P}$ such that $\mu_\mathcal{P}(A) = 0$, then $B \in \mathcal{F}_\mathcal{P}$ for all $B \subseteq A$ [Royden 1968]. We will often also require the sample space to be the full set of memories $\mathsf{Mem}[V]$ for some V. A probability space $\mathcal{P}$ with $\Omega_\mathcal{P} \subseteq \mathsf{Mem}[V]$ can be *extended* as follows: $\mathsf{ext}(\mathcal{P}) \triangleq \langle \mathsf{Mem}[V], \mathcal{F}, \mu \rangle$ where $\mathcal{F} \triangleq \{A \subseteq \mathsf{Mem}[V] \mid A \cap \Omega_\mathcal{P} \in \mathcal{F}_\mathcal{P}\}$ and $\mu(A) \triangleq \mu_\mathcal{P}(A \cap \Omega_\mathcal{P})$.

Projections of probability spaces are defined as $\pi_U(\mathcal{P}) = \langle \Omega, \mathcal{F}, \mu \rangle$, where $\Omega = \pi_U(\Omega_\mathcal{P})$, $\mathcal{F} = \{\pi_U(A) \mid A \in \mathcal{F}_\mathcal{P}\}$, and $\mu(A) = \mu_\mathcal{P}(A * \pi_{\mathsf{Var} \setminus U}(\Omega_\mathcal{P}))$.

3.2.2 Refinement of Probability Spaces

Although probability spaces were historically developed for defining continuous probability distributions, they have more recently also been used to define the amount of information that is available about a probability distribution in the style of intuitionistic logic [Li et al. 2023, 2024]. In intuitionistic logic, the space of models is equipped with a preorder $\preceq$, which defines when some model has more information than another model. Assertions must be stable with respect to $\preceq$, that is, if $m \vDash P$ and $m \preceq m'$, then $m' \vDash P$. Intuitively speaking, as we gain more information about the state of the world, P remains true.

For example, suppose that m is a mapping from variables to real numbers, up to a certain precision. We may have that $m(x) = 3.14\ldots$ and $m_1(x) = 3.140\ldots$ and $m_2(x) = 3.142\ldots$, where the ellipsis indicates the limits of precision in each model. So, $m \preceq m_1$ and $m \preceq m_2$ since both m_1 and m_2 have more precision than m about the value of x. It is therefore neither true that $m \vDash x \leq \pi$ nor $m \vDash \neg(x \leq \pi)$, since each of those assertions is true under one of m_1 or m_2, so their truth under m would not be stable with respect to $\preceq$.

In the case of probability spaces, the information in the preorder $\mathcal{P} \preceq \mathcal{Q}$ can be gained across

two dimensions: by expanding the memory footprint, or by making the event space more fine grained. For $\mathcal{P}$ and $\mathcal{Q}$ such that $\Omega_{\mathcal{P}} \subseteq \mathsf{Mem}[V]$, we define $\mathcal{P} \preceq \mathcal{Q}$ as follows:

Definition 3.4 (Refinement Preorder)**.**

$$\begin{aligned} \mathcal{P} \preceq \mathcal{Q} \quad & \textit{iff} \quad \Omega_{\mathcal{P}} \subseteq \pi_V(\Omega_{\mathcal{Q}}) \\ & \textit{and} \quad \mathcal{F}_{\mathcal{P}} \subseteq \{\pi_V(A) \mid A \in \mathcal{F}_{\mathcal{Q}}\} \\ & \textit{and} \quad \forall A \in \mathcal{F}_{\mathcal{P}}.\ \mu_{\mathcal{P}}(A) = \mu_{\mathcal{Q}}\left(\bigcup \{B \in \mathcal{F}_{\mathcal{Q}} \mid \pi_V(B) = A\}\right) \end{aligned}$$

So, $\mathcal{P} \preceq \mathcal{Q}$ iff $\mathcal{P}$ contains smaller sample and event spaces, but $\mathcal{P}$ and $\mathcal{Q}$ agree on the probability of events whose projections are measurable in $\mathcal{P}$. We now demonstrate the order, returning to example program from Section 3.1.3 $x :\approx \mathbf{Ber}\left(\frac{1}{2}\right) \mathbin{⨟} y := x$. Let $\mathcal{Q}$ be the maximally precise probability space that describes the result of running that program, so $\Omega_{\mathcal{Q}} = \mathsf{Mem}[\{x, y\}]$, $\mathcal{F}_{\mathcal{Q}} = 2^{\Omega_{\mathcal{Q}}}$, and $\mu_{\mathcal{Q}}$ is defined as follows:

$$\mu_{\mathcal{Q}}(A) = \begin{cases} 1 & \text{if } \{[x := 0, y := 0], [x := 1, y := 1]\} \subseteq A \\ \frac{1}{2} & \text{if } [x := 0, y := 0] \in A \text{ xor } [x := 1, y := 1]\} \in A \\ 0 & \text{otherwise} \end{cases}$$

We will now show two different ways to approximate $\mathcal{Q}$. The first approximation $\mathcal{P}_1$ is achieved by eliminating the ability to measure individual values of y, as we did in Section 3.1.3. In that case, $\Omega_{\mathcal{P}_1}$ is still $\mathsf{Mem}[\{x, y\}]$, but now $\mathcal{F}_{\mathcal{P}_1}$ contains only those events where either $y := 0$ and $y := 1$ appear or neither appear.

$$\mathcal{F}_{\mathcal{P}_1} = \{A \subseteq \mathsf{Mem}[\{x, y\}] \mid \forall \sigma \in A.\ \sigma(y) \in \{0, 1\} \Rightarrow \sigma[y := 1 - \sigma(y)] \in A\}$$

Clearly though, both probability spaces do agree on, for example, the probability of the event $x \mapsto 0 * y \in \{0, 1\}$, and so we have that $\mathcal{P}_1 \preceq \mathcal{Q}$.

Another way to approximate $\mathcal{Q}$ is to lose information about y altogether. Let $\mathcal{P}_2$ be a probability space over $\mathsf{Mem}[\{x\}]$, where $\mathcal{F}_{\mathcal{P}_2} = 2^{\mathsf{Mem}[\{x\}]}$ and $\mu_{\mathcal{P}_2}$ assigns probability $\frac{1}{2}$ to the events where $x \mapsto 0$ and $x \mapsto 1$. We also have that $\mathcal{P}_2 \preceq \mathcal{Q}$ since the probabilities of the events only involving x are the same in both $\mathcal{P}_2$ and $\mathcal{Q}$.

Any proper distribution $\mu \in \mathcal{D}(\mathsf{Mem}[U])$, can be used as a probability space where $\Omega_\mu = \mathsf{Mem}[U]$, $\mathcal{F}_\mu = 2^{\mathsf{Mem}[U]}$ is the greatest σ-algebra on $\mathsf{Mem}[U]$, and $\mu_\mu(A) = \sum_{\sigma \in A} \mu(\sigma)$.

3.2.3 Operations on Probability Spaces

Mirroring the logical connectives introduced in Section 3.1, there are operations on probability spaces that semantically correspond to logical outcome assertions $\varphi * \psi$ and $\bigoplus_{X \sim d} \varphi$. The first operation is the independent product of two probability spaces $\mathcal{P} \otimes \mathcal{Q}$, which constructs a new probability space in which the probability of composite events $A * B = \{\sigma \uplus \tau \mid \sigma \in A, \tau \in B\}$ is given by the product of the individual probabilities of $A \in \mathcal{F}_{\mathcal{P}}$ and $B \in \mathcal{F}_{\mathcal{Q}}$.

$$\mu_{\mathcal{P} \otimes \mathcal{Q}}(A * B) = \mu_{\mathcal{P}}(A) \cdot \mu_{\mathcal{Q}}(B)$$

To formally define the independent product, we first need a notion of a product event space $\mathcal{F}_1 \otimes \mathcal{F}_2$, which is the smallest σ-algebra containing the conjunctions of all events in $\mathcal{F}_1$ and $\mathcal{F}_2$:

Definition 3.5 (Product Event Space)**.** *Given two σ-algebras $\mathcal{F}_1 \subseteq 2^{\mathsf{Mem}[U]}$ and $\mathcal{F}_2 \subseteq 2^{\mathsf{Mem}[V]}$ such that $U \cap V = \emptyset$, the product event space $\mathcal{F}_1 \otimes \mathcal{F}_2$ is defined as follows:*

$$\mathcal{F}_1 \otimes \mathcal{F}_2 \triangleq \sigma\left(\{A * B \mid A \in \mathcal{F}_1, B \in \mathcal{F}_2\}\right)$$

The independent product also relies on a lemma due to Bao et al. [2025], which states that any event in $\mathcal{F}_1 \otimes \mathcal{F}_2$ is equal to a disjoint union of conjunctions of events from $\mathcal{F}_1$ and $\mathcal{F}_2$.

Lemma 3.6 (Event Decomposition)**.** *For any probability space $\mathcal{P}_1$ and $\mathcal{P}_2$ over countable sample spaces, there exist countable index sets I and J and disjoint sequences of events $(A_i)_{i \in I}$ and $(B_j)_{j \in J}$ such that for any event $A \in \mathcal{F}_1 \otimes \mathcal{F}_2$, there exist $I' \subseteq I$ and $J' \subseteq J$ such that $A = \biguplus_{(i,j) \in I' \times J'} A_i * B_j$.*

Proof. Follows immediately from Lemma C.5 of Bao et al. [2025]. □

Note that since $\mathsf{Mem}[V]$ is countable for any V, Lemma 3.6 applies to any probability spaces over memories.

Definition 3.7 (Independent Product)**.** *For probability spaces $\mathcal{P}$ and $\mathcal{Q}$ such that $\Omega_{\mathcal{P}} \subseteq \mathsf{Mem}[U]$, $\Omega_{\mathcal{Q}} \subseteq \mathsf{Mem}[V]$, and $U \cap V = \emptyset$, the independent product is defined as:*

$$\mathcal{P} \otimes \mathcal{Q} \triangleq \langle \Omega_{\mathcal{P}} * \Omega_{\mathcal{Q}}, \mathcal{F}_{\mathcal{P}} \otimes \mathcal{F}_{\mathcal{Q}}, \mu \rangle$$

where the probability measure $\mu\colon \mathcal{F}_{\mathcal{P}} \otimes \mathcal{F}_{\mathcal{Q}} \to \mathbb{R}$ *is defined as follows:*

$$\mu(A) \triangleq \sum_{(i,j)\in I\times J} \mu_{\mathcal{P}}(A_i) \cdot \mu_{\mathcal{Q}}(B_j)$$

and the sequences $(A_i)_{i\in I}$ *and* $(B_j)_{j\in J}$ *are those witnessed by Lemma 3.6.*

Next, we define an operation to capture the semantics of the outcome conjunction. Unlike the independent product, which combines two probability spaces over disjoint variable sets, the next operation combines a collection probability spaces over the same variables, but disjoint *sample spaces*. When we sample a fresh random variables $x :\approx d$, the variable x witnesses a partition of the entire sample space, forming the basis for constructing a *direct sum* [Fremlin 2001, 214L].

Definition 3.8 (Direct Sum)**.** *Given a countable index set* I*, discrete distribution* $\xi \in \mathcal{D}(I)$*, and a probability spaces* $\mathcal{P}_i = \langle \Omega_i, \mathcal{F}_i, \mu_i \rangle$ *for each* $i \in I$ *such that all the* Ω_i *are pairwise disjoint, the direct sum is defined as follows:*

$$\bigoplus_{i\sim\xi} \mathcal{P}_i \triangleq \left\langle \biguplus_{i\in I} \Omega_i, \mathcal{F}, \mu \right\rangle$$

where:

$$\mathcal{F} \triangleq \{A \subseteq \Omega \mid \forall i \in I.\, A \cap \Omega_i \in \mathcal{F}_i\} \qquad \mu(A) \triangleq \sum_{i\in I} \xi(i) \cdot \mu_i(A \cap \Omega_i)$$

In a direct sum, the sample space is the disjoint union of all the individual sample spaces, the measurable events are those events whose projections into each Ω_i are measurable according to $\mathcal{F}_i$, and the probability measure is given by a convex sum.

We conclude by discussing some pertinent properties of the two operations above. Both of the operations above are monotone with respect to the refinement order from Section 3.2.2 [Zilberstein et al. 2026, Lemmas B.4 and B.5]:

$$\mathcal{P}_1 \preceq \mathcal{Q}_1 \quad \wedge \quad \mathcal{P}_2 \preceq \mathcal{Q}_2 \qquad \Longrightarrow \qquad \mathcal{P}_1 \otimes \mathcal{P}_2 \preceq \mathcal{Q}_1 \otimes \mathcal{Q}_2$$

$$(\forall i.\mathcal{P}_i \preceq \mathcal{Q}_i) \qquad \Longrightarrow \qquad \bigoplus_{i\sim\xi} \mathcal{P}_i \preceq \bigoplus_{i\sim\xi} \mathcal{Q}_i$$

In addition, independent products distribute over direct sums [Zilberstein et al. 2026, Lemmas B.3], which will form the basis for some of the entailment laws in Section 3.3.3.

$$\left(\bigoplus_{i\sim\xi} \mathcal{P}_i\right) \otimes \mathcal{Q} = \bigoplus_{i\sim\xi} (\mathcal{P}_i \otimes \mathcal{Q})$$

3.3 OUTCOME ASSERTIONS

We can now formally define the syntax and semantics of outcome assertions based on the mathematical preliminaries from the prior section. The syntax of outcome assertions is shown in Figure 3.1. The first five items are typical logical constructs and the final three are specialized assertions for compositional reasoning about probabilistic programs, as introduced in Section 3.1. Throughout the remainder of this section, we will describe the semantics and additional properties of outcome assertions.

3.3.1 Semantics

Outcome assertions are modeled both by a context $\Gamma : \mathsf{LVar} \to \mathsf{Val}$, which assigns values to logical variables, and a probability space $\mathcal{P}$. The semantics is shown in Figure 3.2 in the form of

$\varphi ::= \top$	*(True)*
$\mid \bot$	*(False)*
$\mid \varphi \wedge \psi$	*(Conjunction)*
$\mid \varphi \vee \psi$	*(Disjunction)*
$\mid \exists X.\, \varphi$	*(Existential Quantification)*
$\mid \varphi * \psi$	*(Separating Conjunction)*
$\mid \bigoplus_{X\sim d(E)} \varphi$	*(Outcome Conjunction)*
$\mid \lceil P \rceil$	*(Almost Sure Assertion)*

Figure 3.1: Outcome Assertion syntax.

$$
\begin{array}{ll}
\Gamma, \mathcal{P} \vDash \top & \text{always} \\
\Gamma, \mathcal{P} \vDash \bot & \text{never} \\
\Gamma, \mathcal{P} \vDash \varphi \wedge \psi & \text{iff} \quad \Gamma, \mathcal{P} \vDash \varphi \quad \text{and} \quad \Gamma, \mathcal{P} \vDash \psi \\
\Gamma, \mathcal{P} \vDash \varphi \vee \psi & \text{iff} \quad \Gamma, \mathcal{P} \vDash \varphi \quad \text{or} \quad \Gamma, \mathcal{P} \vDash \psi \\
\Gamma, \mathcal{P} \vDash \exists X.\, \varphi & \text{iff} \quad \Gamma[X := v], \mathcal{P} \vDash \varphi \quad \text{for some} \quad v \in \mathsf{Val} \\
\Gamma, \mathcal{P} \vDash \bigoplus_{X \sim d(E)} \varphi & \text{iff} \quad \forall v.\, \Gamma[X := v], \mathrm{ext}(\mathcal{P}_v) \vDash \varphi \quad \text{and} \quad \bigoplus_{v \sim \mu} \mathcal{P}_v \preceq \mathcal{P} \\
& \qquad \text{for some} \quad (\mathcal{P}_v)_{v \in \mathrm{supp}(\mu)} \quad \text{where} \quad \mu = d(\llbracket E \rrbracket_{\mathsf{LExp}}(\Gamma)) \\
\Gamma, \mathcal{P} \vDash \varphi * \psi & \text{iff} \quad \Gamma, \mathcal{P}_1 \vDash \varphi \ \text{ and } \ \Gamma, \mathcal{P}_2 \vDash \psi \ \text{ for some } \ \mathcal{P}_1, \mathcal{P}_2, \text{such that } \mathcal{P}_1 \otimes \mathcal{P}_2 \preceq \mathcal{P} \\
\Gamma, \mathcal{P} \vDash \lceil P \rceil & \text{iff} \quad (\!|P|\!)^V_\Gamma S \in \mathcal{F}_{\mathcal{P}} \quad \text{and} \quad \mu_{\mathcal{P}}\left((\!|P|\!)^V_\Gamma\right) = 1
\end{array}
$$

Figure 3.2: The satisfaction relation for outcome assertions, where $\Gamma : \mathsf{LVar} \to \mathsf{Var}$ is a logical context and $\mathcal{P} = \langle \mathsf{Mem}[V], \mathcal{F}_{\mathcal{P}}, \mu_{\mathcal{P}} \rangle$ is a complete probability space. All the existentially quantified probability spaces are also complete.

a satisfaction relation $\Gamma, \mathcal{P} \vDash \varphi$. The semantics of the first five constructs is standard; $\top$ is always true, $\bot$ is never true, $\varphi \wedge \psi$ is true iff both φ and ψ are true, $\varphi \vee \psi$ is true iff either φ or ψ is true, and $\exists X.\varphi$ is true iff φ is true with the context updated such that X takes on some value v.

Moving to the probabilistic constructs, the semantics is based on the operations on probability spaces from Section 3.2.3. The outcome conjunction $\bigoplus_{X \sim d(E)} \varphi$ holds iff $\mathcal{P}$ can be decomposed into a direct sum $\bigoplus_{i \sim \mu} \mathcal{P}_i$ such that $\mu = d(\llbracket E \rrbracket_{\mathsf{LExp}}(\Gamma))$ and each $\mathcal{P}_i$ satisfies φ with X set to i.

Similarly, a separating conjunction $\varphi * \psi$ is satisfied if the probability space $\mathcal{P}$ can be decomposed into an independent product $\mathcal{P}_1 \otimes \mathcal{P}_2$ such that $\mathcal{P}_1$ and $\mathcal{P}_2$ satisfy φ and ψ individually. The fact that $\mathcal{P}_1 \otimes \mathcal{P}_2 \preceq \mathcal{P}$ and not $\mathcal{P}_1 \otimes \mathcal{P}_2 = \mathcal{P}$ is crucial to ensure monotonicity of the satisfaction relation. As we saw in Section 3.1, adding information can make events no longer independent, so we may need to choose a decomposition with less information in order to make the assertion valid. For example, $(x \sim \mathbf{Ber}\left(\frac{1}{2}\right)) * \lceil y \in \{0, 1\} \rceil$ is true even if x and y are correlated, but the model where it is true must be constructed using an independent product where the probability space over y cannot measure anything finer than $\lceil y \in \{0, 1\} \rceil$.

Finally, satisfaction of pure assertions $\lceil P \rceil$ requires that P occurs with probability 1. To express this, we use the notation $(\!|P|\!)^V_\Gamma = \{\sigma \in \mathsf{Mem}[V] \mid \Gamma, \sigma \vDash P\}$ from Section 2.4.1. The definition above is an intuitionistic or affine interpretation of separation logic [Docherty 2019], meaning that assertions may not describe the entire space of variables, or alternatively, that information about variables can be dropped, for example:

$$\lceil x \mapsto X * y \mapsto Y \rceil \quad \Longrightarrow \quad \lceil x \mapsto X \rceil$$

To ensure consistency, the satisfaction relation must be monotone.

Theorem 3.9 (Satisfaction Monotonicity)**.** *For any context* Γ*, outcome assertions* φ*, and probability spaces* $\mathcal{P}$ *and* $\mathcal{Q}$*:*

$$\Gamma, \mathcal{P} \vDash \varphi \quad \textit{and} \quad \mathcal{P} \preceq \mathcal{Q} \quad \Longrightarrow \quad \Gamma, \mathcal{Q} \vDash \varphi$$

Proof. By induction on the structure of φ. Most cases follow trivially. For $\lceil P \rceil$ assertions, we use the fact that $\mathcal{Q}$ is complete to ensure that $(\!|P|\!)^V_\Gamma$ is measurable, where $\Omega_\mathcal{Q} = \mathsf{Mem}[V]$. □

We conclude by providing some syntactic sugar for assertions, where in each case X is a fresh variable. As we saw previously, $e \sim d(E)$ expresses that the expression e is distributed according to $d(E)$. In addition $\varphi \oplus_E \psi$ is a binary outcome conjunction and $\varphi \oplus_{\geq E} \psi$ is a bounded outcome conjunction.

$$e \sim d(E) \triangleq \bigoplus_{X \sim d(E)} \lceil e \mapsto X \rceil \qquad \varphi \oplus_E \psi \triangleq \bigoplus_{X \sim \mathbf{Ber}(E)} (\lceil X = 1 \rceil * \varphi) \vee (\lceil X = 0 \rceil * \psi)$$

$$\varphi \oplus_{\geq E} \psi \triangleq \exists X.\, \lceil X \geq E \rceil * (\varphi \oplus_X \psi)$$

3.3.2 Precise and Convex Assertions

Certain properties about outcome assertions dictate how they can be manipulated. The first property is precision, a well-known property from separation logic that serves as a sufficient condition for when the separating conjunction distributes over other connectives [Calcagno et al. 2007; Vafeiadis 2011]. Formally, an assertion is precise if it has a unique smallest model.

Definition 3.10 (Precision). *An assertion φ is* precise *if for any Γ under which φ is satisfiable there is a unique smallest probability space $\mathcal{P}$ such that $\Gamma, \mathcal{P} \vDash \varphi$ and if $\Gamma, \mathcal{P}' \vDash \varphi$, then $\mathcal{P} \preceq \mathcal{P}'$. We write* $\mathsf{precise}(\varphi_1, \ldots, \varphi_n)$ *if φ_i is precise for all $1 \leq i \leq n$.*

In the probabilistic setting, precision intuitively means that every measurable event has an exact probability. For example, $\lceil P \rceil$ is precise for any P, since the measurable event P occurs with probability exactly 1. An outcome conjunction $\bigoplus_{X \sim d(E)} \varphi$ is precise as long as φ is precise and it witnesses a partition of the state space, which we formalize by saying that $\varphi \Rrightarrow \lceil e \mapsto X \rceil$ for some expression e. For example, the assertion $\bigoplus_{X \sim \mathbf{Ber}(p)} \lceil x \mapsto X \rceil$ is precise because $\mathbb{P}[x \mapsto 1] = p$ and $\mathbb{P}[x \mapsto 0] = 1 - p$. On the other hand, $\bigoplus_{X \sim \mathbf{Ber}(p)} \lceil \exists Y.\ x \mapsto Y * Y \leq X \rceil$ is not precise since it does not witness a partition of the sample space and thus $0 \leq \mathbb{P}[x \mapsto 1] \leq p$ and $1 - p \leq \mathbb{P}[x \mapsto 0] \leq 1$ and we could therefore construct many different minimal probability spaces to satisfy the assertion by varying the probabilities of the events $x \mapsto 1$ and $x \mapsto 0$.

Finally, a separating conjunction $\varphi * \psi$ is precise as long as φ and ψ are precise, since we obtain the exact probabilities of each measurable event via multiplication due to the probabilistic independence property. The precision laws are formalized below.

$$\frac{}{\mathsf{precise}(\lceil P \rceil)} \qquad \frac{\mathsf{precise}(\varphi) \qquad \varphi \Rrightarrow \lceil e \mapsto X \rceil}{\mathsf{precise}(\bigoplus_{X \sim d(E)} \varphi)} \qquad \frac{\mathsf{precise}(\varphi, \psi)}{\mathsf{precise}(\varphi * \psi)}$$

When an assertion is imprecise, a weaker property called *convexity* can still hold. Intuitively, φ is convex if $\varphi \oplus_p \varphi \Rrightarrow \varphi$, and it is defined formally below.

Definition 3.11 (Convex Assertions). $\mathsf{convex}(\varphi)$ *iff for all Γ under which φ is satisfiable, there exist Ω, $\mathcal{F}$, and a convex set S of probability measures on $\mathcal{F}$ such that:*

$$\forall \mathcal{P}. \quad \Gamma, \mathcal{P} \vDash \varphi \qquad \textit{iff} \qquad \exists \mu \in S. \quad \langle \Omega, \mathcal{F}, \mu \rangle \preceq \mathcal{P}$$

Any precise assertion is convex since the set S is just a singleton set containing the probability measure witnessed by the definition of precision. The assertion $\varphi \oplus_{\geq E} \psi$ is an example of an assertion that is convex, but not precise, since we only have a bound on the probabilities of φ and

$$\lceil P * Q \rceil \dashv\vdash \lceil P \rceil * \lceil Q \rceil \qquad \frac{P \vdash Q}{\lceil P \rceil \vdash \lceil Q \rceil} \qquad \frac{\varphi \vdash \varphi' \qquad \psi \vdash \psi'}{\varphi * \psi \vdash \varphi' * \psi'} \qquad \frac{\varphi \vdash \psi}{\bigoplus_{X \sim d(E)} \varphi \vdash \bigoplus_{X \sim d(E)} \psi}$$

$$\frac{Y \notin \mathsf{vars}(\varphi)}{\bigoplus_{X \sim d(E)} \varphi \vdash \bigoplus_{Y \sim d(E)} \varphi[Y/X]} \qquad \frac{X \notin \mathsf{vars}(\psi) \quad \mathsf{convex}(\varphi)}{\bigoplus_{X \sim d(E)} \varphi \vdash \varphi}$$

$$\frac{X \notin \mathsf{vars}(\psi)}{(\bigoplus_{X \sim d(E)} \varphi) * \psi \vdash \bigoplus_{X \sim d(E)} (\varphi * \psi)} \qquad \frac{X \notin \mathsf{vars}(\psi) \qquad \varphi \Rightarrow \lceil e \mapsto X \rceil \qquad \mathsf{precise}(\psi)}{\bigoplus_{X \sim d(E)} (\varphi * \psi) \vdash (\bigoplus_{X \sim d(E)} \varphi) * \psi}$$

Figure 3.3: Selected entailment laws, where $\varphi[E/X]$ is a syntactic substitution of E for X in φ.

ψ holding rather than an exact likelihood. A separating conjunction $\varphi * \psi$ is only convex if one of the conjuncts is precise, which rules out hidden correlations between the terms.

$$\frac{\mathsf{precise}(\varphi)}{\mathsf{convex}(\varphi)} \qquad \frac{\mathsf{convex}(\varphi, \psi) \qquad \varphi \Rightarrow \lceil e \mapsto 1 \rceil \qquad \psi \Rightarrow \lceil e \mapsto 0 \rceil}{\mathsf{convex}(\varphi \oplus_{\geq E} \psi)} \qquad \frac{\mathsf{convex}(\varphi) \qquad \mathsf{precise}(\psi)}{\mathsf{convex}(\varphi * \psi)}$$

3.3.3 Entailment Laws

The final element that we need to work with and manipulate outcome assertions is entailment laws. We write $\varphi \vdash \psi$ to mean φ implies ψ is derivable, which semantically means that for any Γ and $\mathcal{P}$, if $\Gamma, \mathcal{P} \vDash \varphi$, then $\Gamma, \mathcal{P} \vDash \psi$.

Some selected entailment laws are shown in Figure 3.3. First, separating conjunctions can be moved inside and outside of almost sure assertions. Implications can be performed inside of almost sure assertions, separating conjunctions, and outcome conjunctions.

Variables bound by outcome conjunctions can be α-renamed as long as the new name is fresh. Outcome conjunctions of convex assertions can be collapsed. Finally, any assertion ψ which does not contain the variable X which is bound by an outcome conjunction can be distributed inside the outcome conjunction, but it can only be factored out of the outcome conjunction if it is precise.

To see why precision is needed in the latter case, consider the assertion:

$$\bigoplus_{X \sim \mathbf{Ber}(p)} \left(\lceil x \mapsto X \rceil * \exists Y.\, y \sim \mathbf{Ber}\,(Y) \right)$$

If the value of Y depends on X, then x and y are correlated, and therefore we cannot factor the information about y out to obtain $(x \sim \mathbf{Ber}(P)) * \exists Y.\ y \sim \mathbf{Ber}(Y)$. However, if y is distributed in some fixed way $\bigoplus_{X \sim \mathbf{Ber}(p)} (\lceil x \mapsto X \rceil * y \sim \mathbf{Ber}(q))$, then no correlation is possible and it can thus be factored out to obtain $(x \sim \mathbf{Ber}(p)) * (y \sim \mathbf{Ber}(q))$.

3.4 RELATED WORK

The idea to model probabilistic independence with separation was first explored by Barthe et al. [2020] with their introduction of Probabilistic Separation Logic (PSL). However, PSL was limited both in its ability to reason about the granularity of independence and also about conditional independence. Dependence and Independence Bunched Implications (DIBI) later extended PSL with a conditioning operator, for improved abilities to reason about control flow [Bao et al. 2021].

Li et al. [2023, 2024] later introduced the Lilac logic, which reformulated probabilistic separation based on measure theory in order express independence at varying levels of granularity, as we saw in Section 3.1.3. Lilac also features a conditioning modality, which supports reasoning about continuous probability, but lacks some of the compositional reasoning properties of the direct sum, which will become important in the next chapter. On the other hand, Lilac's conditioning modality does support reasoning about continuous probability distributions, so both approaches have relative strengths. Bluebell is based on a similar construction as Lilac, but is limited to discrete distributions and additionally offers relational reasoning about many program executions [Bao et al. 2025].

The definition of separation in this thesis draws both from PSL [Barthe et al. 2020] and also newer logics. We use probability spaces for reasoning about independence at varying levels of granularity, but also use a cartesian product construction to fully separate memory footprints rather than the more general approach of Lilac and Bluebell, which rely on a theorem stating that independent products are unique [Li et al. 2023, Lemma 2.3]. We use the explicit product construction in order to guarantee that each variable can only occur on one side of the $*$, making

mutation rules simpler. Lilac does not allow mutable state, and so mutation is not a factor. On the other hand, Bluebell handles mutation by explicitly tracking permissions, but the product construction is simpler to use than the permission approach.

The notion of *precision* has been previously studied in separation logics, in part to explain when the separating conjunction distributes over other logical connectives, such as the regular conjunction [Calcagno et al. 2007; Vafeiadis 2011]. In PSL, precision (under the name *supported*) was used to ensure that the guard of an if statement remains independent of the postcondition of the two branches. Conditioning à la Lilac and Bluebell provides a more flexible way to reason about control flow without forcing the guard to be independent of the states in the two branches.

CHAPTER 4

PROBABILISTIC OUTCOME LOGIC

Having just formalized an assertion language for specifying the outcomes of a probabilistic program, we can now use that assertion language to define Probabilistic Outcome Logic (pOL). Although prior logics enable compositional reasoning about probabilistic programs using probabilistic independence [Barthe et al. 2020; Bao et al. 2021, 2025; Li et al. 2023], all of those logics have limited abilities to handle control flow and loops, whereas pOL is more powerful in those regards (refer to Section 4.5 for a more complete comparison). Therefore, pOL provides an expressive foundation on top of which we add the ability to additionally handle nondeterminism and concurrency in chapters Chapters 5 and 7, respectively.

4.1 OVERVIEW

Logics for expressive and compositional reasoning about probabilistic programs require a combination of several features. Expressiveness stems from describing distributions of outcomes. Compositionality relies on the ability to reason locally about state, à la separation logic, and also to reason locally about random outcomes. To that end, pOL augments standard Outcome Logic [Zilberstein 2025] with the probabilistic separation capabilities from the previous chapter.

As in prior program logics, pOL expresses specifications as triples $\langle \varphi \rangle\ C\ \langle \psi \rangle$, where the precondition φ and postcondition ψ are outcome assertions (Figure 3.1) and $C \in cmd_{\mathrm{P}}$ is a pCL program whose syntax and semantics were given in Section 2.2. As a simple example, we specify a fair coin flip using a postcondition that states that x is distributed according to a coin flip.

$$\langle \lceil x \mapsto - \rceil \rangle\ x :\approx \mathbf{Ber}\left(\tfrac{1}{2}\right)\ \langle x \sim \mathbf{Ber}\left(\tfrac{1}{2}\right) \rangle$$

However, deriving specifications for larger composite programs requires rules for reasoning in a modular way. We will now describe a few of the modular reasoning principles of pOL, and in later sections we will formalize those principles as sounds inference rules.

4.1.1 Local Reasoning and the Frame Rule

As we saw in Section 2.4, the Frame rule enables local reasoning by lifting specifications about a local *heaplet* into a larger global context [Reynolds 2002; O'Hearn et al. 2001; Yang and O'Hearn 2002]. In the probabilistic setting, the Frame rule takes the same form, but additionally guarantees that the local variables are distributed independently from other variables [Barthe et al. 2020].

$$\frac{\langle \varphi \rangle\ C\ \langle \psi \rangle}{\langle \varphi * \vartheta \rangle\ C\ \langle \psi * \vartheta \rangle}\textsc{Frame}$$

For instance, returning to the example from the beginning of the section, we can use the Frame rule to prove that a freshly sampled variable is probabilistically independent from other variables.

$$\frac{\langle \lceil x \mapsto - \rceil \rangle\ x :\approx \mathbf{Ber}\left(\tfrac{1}{2}\right)\ \langle x \sim \mathbf{Ber}\left(\tfrac{1}{2}\right) \rangle}{\langle \lceil x \mapsto - \rceil * \left(y \sim \mathbf{Ber}\left(\tfrac{1}{2}\right)\right) \rangle\ x :\approx \mathbf{Ber}\left(\tfrac{1}{2}\right)\ \langle \left(x \sim \mathbf{Ber}\left(\tfrac{1}{2}\right)\right) * \left(y \sim \mathbf{Ber}\left(\tfrac{1}{2}\right)\right) \rangle}\textsc{Frame}$$

The independence information is very useful; it allows us to obtain the entire joint distribution over all of the program variables.

4.1.2 Conditioning and Control Flow

Consider the following program, which simulates a three-sided die using only Bernoulli distributions. Upon termination, x will be a uniformly distributed integer between 1 and 3.

$$\begin{array}{l} x :\approx \mathbf{Ber}\left(\tfrac{1}{3}\right) \fatsemi \\ \textbf{if}\ x = 1\ \textbf{then} \\ \quad \textbf{skip} \\ \textbf{else} \\ \quad y :\approx \mathbf{Ber}\left(\tfrac{1}{2}\right) \fatsemi \\ \quad x \coloneqq 2 + y \end{array}$$

Since the control flow of the program depends on random behavior, proving its correctness will require us to know how likely the program is to enter each branch of the if statement. Therefore, although it would be sound to adapt the standard If rule of Hoare Logic to OL, that rule is not sufficiently expressive to reason about probabilistic control flow.

$$\frac{\langle \lceil P * b \mapsto 1 \rceil \rangle\ C_1\ \langle \lceil Q \rceil \rangle \qquad \langle \lceil P * b \mapsto 0 \rceil \rangle\ C_2\ \langle \lceil Q \rceil \rangle}{\langle \lceil P * b \mapsto - \rceil \rangle\ \textbf{if}\ b\ \textbf{then}\ C_1\ \textbf{else}\ C_2\ \langle \lceil Q \rceil \rangle}\textsc{If (Hoare)}$$

Instead, we use a series of rules, which allow us to reason about both the likelihood and behavior of each branch. We show simplified versions of those rules here. First, the Split rule allows us to reason underneath an outcome conjunction.

$$\frac{\langle \varphi \rangle\ C\ \langle \psi \rangle}{\langle \bigoplus_{X \sim d(E)} \varphi \rangle\ C\ \langle \bigoplus_{X \sim d(E)} \psi \rangle}\textsc{Split}$$

In the conclusion, X is bound by the outcome conjunction, but in the premise it is implicitly universally quantified. So, if $\langle \varphi \rangle\ C\ \langle \psi \rangle$ holds for any fixed value of X, then we can conclude that the same pre- and postcondition pair holds underneath the outcome conjunctions. For example, if we initially know that x is distributed according to $\mathbf{Ber}\left(\frac{1}{2}\right)$, and the program increments x, then the Split rule essentially allows us to map the plus 1 function over the distribution of outcomes.

$$\frac{\langle \lceil x \mapsto X \rceil \rangle\ x := x + 1\ \langle \lceil x \mapsto X + 1 \rceil \rangle}{\langle \bigoplus_{X \sim \mathbf{Ber}\left(\frac{1}{2}\right)} \lceil x \mapsto X \rceil \rangle\ x := x + 1\ \langle \bigoplus_{X \sim \mathbf{Ber}\left(\frac{1}{2}\right)} \lceil x \mapsto X + 1 \rceil \rangle}\textsc{Split}$$

Returning to analyzing control flow, if the initial outcome conjunction gives the distribution over values of the test b, then Split puts us in a scenario where the program will deterministically take one branch or the other.

$$\frac{\langle \varphi * \lceil b \mapsto X \rceil \rangle\ \texttt{if}\ b\ \texttt{then}\ C_1\ \texttt{else}\ C_2\ \langle \psi \rangle}{\langle \bigoplus_{X \sim \mathbf{Ber}(p)} \varphi * \lceil b \mapsto X \rceil \rangle\ \texttt{if}\ b\ \texttt{then}\ C_1\ \texttt{else}\ C_2\ \langle \bigoplus_{X \sim \mathbf{Ber}(p)} \psi \rangle}\textsc{Split}$$

Since b is a Boolean-valued test expression, then X in the premise of the above derivation can only take on the values 0 or 1. So, by considering both possible values of X, we force the execution to deterministically take the true or false branch, and then we can use the following two rules to complete the proof.

$$\frac{\langle \varphi * \lceil b \mapsto 1 \rceil \rangle\ C_1\ \langle \psi \rangle}{\langle \varphi * \lceil b \mapsto 1 \rceil \rangle\ \texttt{if}\ b\ \texttt{then}\ C_1\ \texttt{else}\ C_2\ \langle \psi \rangle}\textsc{IfT} \qquad \frac{\langle \varphi * \lceil b \mapsto 0 \rceil \rangle\ C_2\ \langle \psi \rangle}{\langle \varphi * \lceil b \mapsto 0 \rceil \rangle\ \texttt{if}\ b\ \texttt{then}\ C_1\ \texttt{else}\ C_2\ \langle \psi \rangle}\textsc{IfF}$$

In fact, we can derive a composite rule, which is similar to the classic one from Hoare Logic, but which now takes the probabilistic behavior into account.

$$\frac{\langle \varphi * \lceil b \mapsto 1 \rceil \rangle\ C_1\ \langle \psi \rangle \qquad \langle \varphi * \lceil b \mapsto 0 \rceil \rangle\ C_2\ \langle \psi \rangle}{\left\langle \bigoplus_{X \sim \mathbf{Ber}(p)} \varphi * \lceil b \mapsto X \rceil \right\rangle\ \mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \left\langle \bigoplus_{X \sim \mathbf{Ber}(p)} \psi \right\rangle}\ \textsc{If}$$

Returning to the original program, we can use the postcondition:

$$\psi \triangleq \lceil X = 1 * x \mapsto 1 \rceil \vee \left(\lceil X = 0 \rceil * \bigoplus_{Y \sim \mathbf{Ber}\left(\frac{1}{2}\right)} \lceil x \mapsto 2 + Y \rceil \right)$$

It is relatively easy to see that:

$$\bigoplus_{X \sim \mathbf{Ber}\left(\frac{1}{3}\right)} \psi \implies x \sim \mathbf{unif}\ (\{1, 2, 3\})$$

Therefore, using the If rule presented above, we can obtain the following final specification.

$$\left\langle \bigoplus_{X \sim \mathbf{Ber}\left(\frac{1}{3}\right)} \lceil x \mapsto X \rceil \right\rangle\ \mathbf{if}\ x = 1\ \mathbf{then}\ \mathbf{skip}\ \mathbf{else}\ y :\approx \mathbf{Ber}\left(\tfrac{1}{2}\right) \fatsemi x := 2 + y\ \left\langle x \sim \mathbf{unif}\ (\{1, 2, 3\}) \right\rangle$$

In Outcome Logic, compositional reasoning via the Split rule in the way shown above is commonly needed to reason about the control flow in programs, which we will see repeatedly throughout the remaining chapters.

4.1.3 Loops and Almost Sure Termination

The final form of reasoning needed to analyze probabilistic programs consists of rules to modularly reason about loops. We are primarily concerned with *almost sure termination*, the property that a program terminates with probability 1 in the limit. Inspired by Total Hoare Logic [Manna and Pnueli 1974], we can use a ranking function to ensure that the program moves toward termination with each iteration of the loop.

However, unlike standard programs, probabilistic loops need not *always* move toward termination, but rather only must move toward termination with some probability. For example, the

following program terminates with probability $\frac{1}{2}$ on each iteration. So the probability of nontermination after n iterations is $\frac{1}{2^n}$, which clearly goes to zero as n goes to infinity.

$$\textbf{while } x = 0 \textbf{ do } x :\approx \textbf{Ber}\left(\tfrac{1}{2}\right)$$

But the situation can get more complicated. For example, the program may not immediately exit on each iteration, but rather only probabilistically move toward termination, as is the case below.

$$\textbf{while } x > 0 \textbf{ do } x \coloneqq x - 1 \oplus_{\frac{1}{2}} \textbf{skip}$$

We will see how to formally prove almost sure termination for the program above in Section 4.3.3, and sketch the proof informally here. First, we need a ranking function, which in this case is just the value of the variable x. Clearly, when x reaches 0, the program terminates, so all we need to do is show that the rank decreases with some nonzero probability p on each iteration. Clearly, the rank decreases with probability $\frac{1}{2}$ each iteration, so the program will eventually terminate with probability 1. There are many more complex rules for proving termination of probabilistic programs [McIver and Morgan 2005; McIver et al. 2018; Majumdar and Sathiyanarayana 2025], but in this thesis the style above, known as bounded rank, will suffice.

4.2 FORMALIZING OUTCOME TRIPLES

Triples in pOL $\langle \varphi \rangle\ C\ \langle \psi \rangle$ are judgements, where the pre- and postconditions φ and ψ are outcome assertions that we formalized in Chapter 3 and $C \in cmd_{\text{P}}$ is a probabilistic program with syntax and semantics given in Section 2.2. Informally, the meaning of $\langle \varphi \rangle\ C\ \langle \psi \rangle$ is that if the program C is run on a distribution of states described by φ, then the outcomes will be distributed according to ψ. The meaning is similar to Hoare triples $\{P\}\ C\ \{Q\}$, but lifted into a setting where φ, C, and ψ are all probabilistic.

As we saw in Section 4.1.1, pOL triples are also *frame preserving* in that the validity of $\langle \varphi \rangle\ C\ \langle \psi \rangle$ implies the validity of $\langle \varphi * \vartheta \rangle\ C\ \langle \psi * \vartheta \rangle$. Following many higher order separation logics [Birkedal and Yang 2007] including Iris [Jung et al. 2018], the frame property is baked into

the validity definition of the triples. Doing so eliminates the need for cumbersome side conditions that existed in Probabilistic Separation Logic [Barthe et al. 2020].

We write $\vDash^{\text{pOL}} \langle \varphi \rangle\ C\ \langle \psi \rangle$ to mean that a triple is semantically valid in pOL. Validity requires the initial state to be captured by a logical variable context $\Gamma : \mathsf{LVar} \to \mathsf{Val}$, a local probability space $\mathcal{P}$, a frame probability space $\mathcal{P}_F$, and a distribution over states μ. Since $\mathcal{P}$ models the local state, then it must satisfy the precondition, *i.e.,* $\Gamma, \mathcal{P} \vDash \varphi$. The second probability space $\mathcal{P}_F$ models any distribution of variables that are not in the local memory, which will be unchanged after the program executes. Finally, μ is the concrete distribution over memories, and it refines the independent product of $\mathcal{P}$ and $\mathcal{P}_F$, *i.e.,* $\mathcal{P} \otimes \mathcal{P}_F \preceq \mu$.

As long as those conditions are met, then $\vDash^{\text{pOL}} \langle \varphi \rangle\ C\ \langle \psi \rangle$ means that there must be some final probability space $\mathcal{Q}$ such that $[\![C]\!]^{\dagger}(\mu)$ refines $\mathcal{Q} \otimes \mathcal{P}_F$ and $\Gamma, \mathcal{Q} \vDash \psi$. The formal definition of pOL triples is given below.

Definition 4.1 (Probabilistic Outcome Triples). $\vDash^{\text{pOL}} \langle \varphi \rangle\ C\ \langle \psi \rangle$ *iff for any* Γ, μ, *and probability spaces* $\mathcal{P}$ *and* $\mathcal{P}_F$ *such that* $\mathcal{P} \otimes \mathcal{P}_F \preceq \mu$ *and* $\Gamma, \mathcal{P} \vDash \varphi$, *then:*

$$\exists \mathcal{Q}.\quad \mathcal{Q} \otimes \mathcal{P}_F \preceq [\![C]\!]^{\dagger}_{\mathcal{D}}(\mu) \text{ and } \Gamma, \mathcal{Q} \vDash \psi$$

Note that probability spaces range over memories (without $\bot$), so the fact that $\mathcal{Q} \otimes \mathcal{P}_F \preceq [\![C]\!]^{\dagger}(\mu)$ means that $[\![C]\!]^{\dagger}(\mu)(\bot) = 0$, *i.e.,* that C almost surely terminated. In that sense, pOL is similar to a total correctness logic [Manna and Pnueli 1974], where the validity of any triple implies termination. Accordingly, the loop rule will establish termination. For the remainder of the chapter, we will omit the pOL superscript and simply write $\vDash \langle \varphi \rangle\ C\ \langle \psi \rangle$.

4.3 PROOF RULES

Given the formal meaning of pOL triples, we can now develop proof rules to mechanically derive valid judgements in the logic. We write $\vdash \langle \varphi \rangle\ C\ \langle \psi \rangle$ to mean that the triple is *derivable* via the rules. All of the rules in the following sections are sound with respect to the validity definition

$$\frac{}{\langle\varphi\rangle\ \mathbf{skip}\ \langle\varphi\rangle}\textsc{Skip} \qquad \frac{\langle\varphi\rangle\ C_1\ \langle\vartheta\rangle \qquad \langle\vartheta\rangle\ C_2\ \langle\psi\rangle}{\langle\varphi\rangle\ C_1 \fatsemi C_2\ \langle\psi\rangle}\textsc{Seq}$$

$$\frac{\langle\varphi\rangle\ C_1\ \langle\psi_1\rangle \quad \langle\varphi\rangle\ C_2\ \langle\psi_2\rangle \quad \varphi \Rightarrow \lceil e \mapsto E\rceil \quad \psi_1 \Rightarrow \lceil e' \mapsto 1\rceil \quad \psi_2 \Rightarrow \lceil e' \mapsto 0\rceil}{\langle\varphi\rangle\ C_1 \oplus_e C_2\ \langle\psi_1 \oplus_E \psi_2\rangle}\textsc{Prob}$$

$$\frac{\varphi \Rightarrow \lceil b \mapsto 1\rceil \qquad \langle\varphi\rangle\ C_1\ \langle\psi\rangle}{\langle\varphi\rangle\ \mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \langle\psi\rangle}\textsc{IfT} \qquad \frac{\varphi \Rightarrow \lceil b \mapsto 0\rceil \qquad \langle\varphi\rangle\ C_2\ \langle\psi\rangle}{\langle\varphi\rangle\ \mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \langle\psi\rangle}\textsc{IfF}$$

$$\frac{\varphi \Rightarrow \lceil e \mapsto E\rceil \wedge (\psi * \lceil \mathsf{own}(x)\rceil)}{\langle\varphi\rangle\ x := e\ \langle\lceil x \mapsto E\rceil * \psi\rangle}\textsc{Assign} \qquad \frac{\varphi \Rightarrow \lceil e \mapsto E\rceil \wedge (\psi * \lceil \mathsf{own}(x)\rceil)}{\langle\varphi\rangle\ x :\approx d(e)\ \langle(x \sim d(E)) * \psi\rangle}\textsc{Samp}$$

Figure 4.1: Rules for sequential commands in pOL.

(Definition 4.1), which means that any derivable triple is semantically valid.

$$\vdash \langle\varphi\rangle\ C\ \langle\psi\rangle \qquad \Longrightarrow \qquad \vDash \langle\varphi\rangle\ C\ \langle\psi\rangle$$

We will defer the details of the soundness proof to Section 7.4.

4.3.1 Basic Sequential Commands

We begin in Figure 4.1 with the standard rules for analyzing sequential (non-looping) commands. The Skip rule preserves the precondition for a no-op command and the Seq rule compositionally builds a specification for a sequential composition $C_1 \fatsemi C_2$ from derivations for the two subcommands. Soundness of the Seq rule follows from the associativity law of monads.

To analyze a probabilistic choice command $C_1 \oplus_e C_2$, the precondition φ must dictate the value of the probability e, *i.e.*, $\varphi \Rightarrow \lceil e \mapsto E\rceil$. In addition, the postconditions of the two branches ψ_1 and ψ_2 must witness a partition of the program states via the expression e', *i.e.*, $\psi_1 \Rightarrow \lceil e' \mapsto 1\rceil$ and $\psi_2 \Rightarrow \lceil e' \mapsto 0\rceil$. Then, the postcondition of the triple as a whole is simply $\psi_1 \oplus_E \psi_2$.

Next, IfT and IfF are used to create specifications for the true and false branches of an if statement, respectively. Compared to the simpler versions from Section 4.1.2, the ones in Figure 4.1 have a more flexible precondition, only requiring, *e.g.*, that $\varphi \Rightarrow \lceil b \mapsto 1\rceil$ rather than forcing the

precondition to have the form $\varphi * \lceil b \mapsto 1 \rceil$. The more general form allows the guard b to reference multiple variables or probabilistic information, for example if the guard b is the expression $x + y \geq 0$ and the precondition is $(x \sim \mathbf{Ber}(p)) * (y \sim \mathbf{Ber}(q))$, then the condition is clearly met.

$$(x \sim \mathbf{Ber}(p)) * (y \sim \mathbf{Ber}(q)) \Rrightarrow \lceil (x + y \geq 0) \mapsto 1 \rceil$$

Finally, we have two rules for reasoning about the assignment $x := e$ and sampling $x :\approx d(e)$ actions. In both cases, the precondition φ must imply both that the program expression e evaluates to the logical expression E ($\lceil e \mapsto E \rceil$), and that we have permission to write to x ($\mathsf{own}(x)$). In addition, the leftover state ψ about the variables other than x must be separate from x itself in order to properly represent the postcondition. The rules are formulated in that way so as to capture many different modes of use. For example, the expression e may be derived from multiple variables other than just x:

$$\frac{\lceil x \mapsto 1 * y \mapsto 2 \rceil \Rrightarrow \lceil (x + y) \mapsto 3 \rceil \land (\lceil y \mapsto 2 \rceil * \lceil \mathsf{own}(x) \rceil)}{\langle \lceil x \mapsto 1 * y \mapsto 2 \rceil \rangle\ x := x + y\ \langle \lceil x \mapsto 3 \rceil * \lceil y \mapsto 2 \rceil \rangle}\ \textsc{Assign}$$

Alternatively, x itself may be distributed in a complex way:

$$\frac{(x \sim d) * \lceil y \mapsto 2 \rceil \Rrightarrow \lceil y \mapsto 2 \rceil \land (\lceil y \mapsto 2 \rceil * \lceil \mathsf{own}(x) \rceil)}{\langle (x \sim d) * \lceil y \mapsto 2 \rceil \rangle\ x := y\ \langle \lceil x \mapsto 2 \rceil * \lceil y \mapsto 2 \rceil \rangle}\ \textsc{Assign}$$

4.3.2 Structural Rules

Our next set of rules, shown in Figure 4.2, are structural rules that do not depend on the program syntax. As described in Section 4.1.1, the Frame rule allows a local probabilistic specification to be lifted into a larger distribution over memories. The rule of Consequence allows for strengthening of the precondition and weakening of the postcondition. It can be used in conjunction with the entailment laws from Figure 3.3.

The final two rules are for case analysis on outcomes, as described in Section 4.1.2. We include two variants. The first rule, Split1, requires that ψ dictates the partition of the probability spaces in order to construct a final direct sum. The partition is witnessed in a similar fashion to rules

$$\frac{\langle \varphi \rangle\ C\ \langle \psi \rangle}{\langle \varphi * \vartheta \rangle\ C\ \langle \psi * \vartheta \rangle}\textsc{Frame} \qquad \frac{\varphi' \Rightarrow \varphi \qquad \langle \varphi \rangle\ C\ \langle \psi \rangle \qquad \psi \Rightarrow \psi'}{\langle \varphi' \rangle\ C\ \langle \psi' \rangle}\textsc{Consequence}$$

$$\frac{\langle \varphi \rangle\ C\ \langle \psi \rangle \qquad \psi \Rightarrow \lceil e \mapsto X \rceil}{\langle \bigoplus_{X \sim d(E)} \varphi \rangle\ C\ \langle \bigoplus_{X \sim d(E)} \psi \rangle}\textsc{Split1} \qquad \frac{\langle \varphi \rangle\ C\ \langle \psi \rangle \qquad \mathsf{convex}(\psi) \qquad X \notin \mathsf{vars}(\psi)}{\langle \bigoplus_{X \sim d(E)} \varphi \rangle\ C\ \langle \psi \rangle}\textsc{Split2}$$

Figure 4.2: Structural pOL rules.

for establishing precision that we saw in Section 3.3.2, by requiring that $\psi \Rightarrow \lceil e \mapsto X \rceil$ for some expression e. Since X takes on distinct values in each case of the direct sum, then $e \mapsto X$ witnesses that the sample space can be partitioned. If ψ does not witness a partition, then the Split2 rule can instead be used, which requires ψ to be convex and not dependent on X.

We now demonstrate the two modes of use. Below, x is initially distributed according to some distribution $d(E)$, and then it is incremented. The sample space is still partitioned after the increment, which is witnessed by $\lceil x - 1 \mapsto X \rceil$, so Split1 can be used.

$$\frac{\vdash \langle \lceil x \mapsto X \rceil \rangle\ x := x + 1\ \langle \lceil x \mapsto X + 1 \rceil \rangle \qquad \lceil x \mapsto X + 1 \rceil \Rightarrow \lceil x - 1 \mapsto X \rceil}{\vdash \langle \bigoplus_{X \sim d(E)} \lceil x \mapsto X \rceil \rangle\ x := x + 1\ \langle \bigoplus_{X \sim d(E)} \lceil x \mapsto X + 1 \rceil \rangle}\textsc{Split1}$$

On the other hand, the next program samples y, and the resulting sample space is not partitioned according to X. Instead, Split2 is used since the postcondition is convex.

$$\frac{\vdash \langle \lceil x \mapsto X * X \in \{0,1\} \rceil \rangle\ y :\approx \mathbf{Ber}\left(\tfrac{x}{2}\right)\ \langle \exists Y.\ \lceil Y \leq \tfrac{1}{2} \rceil * y \sim \mathbf{Ber}\,(Y) \rangle}{\vdash \langle \bigoplus_{X \sim \mathbf{Ber}\left(\frac{1}{2}\right)} \lceil x \mapsto X * X \in \{0,1\} \rceil \rangle\ y :\approx \mathbf{Ber}\left(\tfrac{x}{2}\right)\ \langle \exists Y.\ \lceil Y \leq \tfrac{1}{2} \rceil * y \sim \mathbf{Ber}\,(Y) \rangle}\textsc{Split2}$$

Another use for the splitting rules is to derive a two-sided rule for if statements. The If rule shown below is derived by first using Split1 to condition on the value of the test b, and then applying IfT and IfF to handle the two cases.

$$\frac{I \vdash_m \langle \varphi * \lceil X = 1 \rceil \rangle\ C_1\ \langle \psi \rangle \qquad I \vdash_m \langle \varphi * \lceil X = 0 \rceil \rangle\ C_2\ \langle \psi \rangle \qquad \varphi \Rightarrow \lceil b \mapsto X \rceil \qquad \psi \Rightarrow \lceil b \mapsto X \rceil}{I \vdash_m \langle \bigoplus_{X \sim \mathbf{Ber}(p)} \varphi \rangle\ \mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \langle \bigoplus_{X \sim \mathbf{Ber}(p)} \psi \rangle}\textsc{If}$$

4.3.3 Analyzing Loops

As we saw in Section 4.1.3, analyzing probabilistic loops and proving almost sure termination is more complicated than loop analysis in deterministic programs. In this section, we present a proof rule for analyzing probabilistic loops. The approach to establishing almost sure termination is based on prior work [McIver and Morgan 2005; Kaminski 2019; McIver et al. 2018]. However, unlike prior work, the pOL loop rule also specifies the distribution of outcomes that occur upon termination, rather than just establishing that the loop terminates.

The basic strategy to ensure termination uses a bounded integer rank $\ell \leq R \leq h$, that strictly decreases with probability $p > 0$ on each iteration. Once the rank reaches the lower bound ℓ, the loop must terminate. That means that at any point, the loop will terminate within $h - \ell$ iterations with probability $p^{h-\ell}$, which is nonzero. Let t be the probability of termination, then we can lower bound t as the the probability of termination within $h - \ell$ iterations plus the probability that the program does not terminate in that time plus t.

$$t \geq p^{h-\ell} + (1 - p^{h-\ell}) \cdot t$$

By subtracting $(1 - p^{h-\ell}) \cdot t$ from both sides of the inequality, we get that $t \cdot p^{h-\ell} \geq p^{h-\ell}$, and since $p > 0$, the only solution to that inequality is $t = 1$. Refer to McIver and Morgan [2005, §2.6] for a more complete discussion.

Our loop rule, shown in Figure 4.3, is built around a loop invariant φ, which is an outcome assertion that may describe a distribution of many different outcomes. The loop invariant must imply that the rank is in bounds, *i.e.,* that $\ell \leq R \leq h$. It must also be the case that when $R = \ell$, φ implies that the loop guard b is false, and otherwise if $R > \ell$ then b is true.

The premise of the rule requires that as long as the invariant φ holds and $R = N > \ell$, then running the loop body C will result in a distribution where the invariant φ still holds, and $R < N$ with probability $p > 0$ and otherwise $R \geq N$. In the case that $R < N$, then R is distributed according to some distribution $d_1(E_1)$ whose support contains only integers between ℓ and $N - 1$.

$$\frac{\langle \varphi * \lceil R = N \rceil \rangle\ C\ \langle \left(\bigoplus_{R \sim d_1(E_1)} \varphi\right) \oplus_p \left(\bigoplus_{R \sim d_2(E_2)} \varphi\right) \rangle \quad \begin{array}{c} \varphi \Rightarrow \lceil \ell \le R \le h \rceil \\ \varphi * \lceil R = \ell \rceil \Rightarrow \lceil b \mapsto 0 \rceil \\ \varphi * \lceil R > \ell \rceil \Rightarrow \lceil b \mapsto 1 \rceil \\ \mathrm{supp}(d_1(E_1)) \subseteq [\ell, N-1] \\ \mathrm{supp}(d_2(E_2)) \subseteq [N, h] \end{array} \quad \begin{array}{c} N > \ell \\ N \notin \mathrm{vars}(\varphi) \\ 0 < p \le 1 \\ \mathrm{precise}(\varphi[\ell/R]) \end{array}}{\langle \bigoplus_{R \sim d(E)} \varphi \rangle\ \textbf{while}\ b\ \textbf{do}\ C\ \langle \varphi[\ell/R] \rangle}\ \textsc{BoundedRank}$$

Figure 4.3: A loop rule for almost-sure termination in pOL.

Similarly, R is distributed according to another distribution $d_2(E_2)$ whose support is between N and h in the case where $R \geq N$.

If those conditions hold, then as long as the initial states satisfy the invariant φ, with R distributed according to any distribution $d(E)$, then the loop will almost surely terminate with the outcomes distributed according to φ with $R = \ell$. We will see examples using BoundedRank in Section 4.4. Additional strategies for proving almost-sure termination are discussed in Section 4.5.

4.4 EXAMPLES

We now introduce two running examples to demonstrate how pOL provides an expressive foundation for reasoning about probabilistic programs with unbounded loops. In future chapters, we will extend the examples with nondeterminism and concurrency, and see how the Outcome Logic proof strategies evolve to deal with increasingly complex combinations of computational effects.

4.4.1 Random Walk

The first example is a random walk in which an agent moves toward the origin with probability $\frac{1}{2}$ on each round. The agent's position is stored in the variable x and the walk terminates when the agent reaches the origin, *i.e.,* $x = 0$. A program implementing the walk is shown below.

$$\begin{array}{l} \textbf{while}\ x > 0\ \textbf{do} \\ \quad x \coloneqq x - 1 \oplus_{\frac{1}{2}} \textbf{skip} \end{array}$$

We want to prove both that the random walk almost surely terminates, and that the agent ends up at position 0. Since pOL triples imply almost sure termination, both requirements are captured

$$\dfrac{\dfrac{}{\langle\lceil x \mapsto N\rceil\rangle\ x \coloneqq x-1\ \langle\lceil x \mapsto N-1\rceil\rangle}\ \textsc{Assign} \qquad \dfrac{}{\langle\lceil x \mapsto N\rceil\rangle\ \mathbf{skip}\ \langle\lceil x \mapsto N\rceil\rangle}\ \textsc{Skip}}{\langle\lceil x \mapsto N*0<N\leq X\rceil\rangle\ x \coloneqq x=1 \oplus_{\frac{1}{2}} \mathbf{skip}\ \langle\lceil x \mapsto N-1\rceil \oplus_{\frac{1}{2}} \lceil x \mapsto N\rceil\rangle}\ \textsc{Prob}$$
$$\dfrac{}{\langle\lceil x \mapsto X*0\leq X\rceil\rangle\ \mathbf{while}\ x>0\ \mathbf{do}\ x \coloneqq x=1 \oplus_{\frac{1}{2}} \mathbf{skip}\ \langle\lceil x \mapsto 0\rceil\rangle}\ \textsc{BoundedRank}$$

Figure 4.4: Almost sure termination proof for a simple random walk in pOL.

by the postcondition $\lceil x \mapsto 0\rceil$. As a precondition, we will use $\lceil x \mapsto X*0\leq X\rceil$, *i.e.,* the agent begins at some arbitrary positive position X. In order to apply BoundedRank, we first need an invariant. The invariant simply states that $x \mapsto R$, the current position x is also the current rank R, and the rank is bounded between 0 and X, the initial starting point.

$$\varphi \triangleq \lceil x \mapsto R*0\leq R\leq X\rceil$$

The other parameters of BoundedRank are instantiated as follows:

$$p \triangleq \frac{1}{2} \qquad d(E) \triangleq \mathbf{unif}\,(\{X\}) \qquad d_1(E_1) \triangleq \mathbf{unif}\,(\{N-1\}) \qquad d_1(E_1) \triangleq \mathbf{unif}\,(\{N\})$$

The rank decreases each iteration with probability $p=\frac{1}{2}$ and all three distributions $d(E)$, $d_1(E_1)$, and $d_2(E_2)$ are uniform distributions over singleton sets, meaning that the outcome conjunctions collapse into a single term. The derivation is shown in Figure 4.4. First, the BoundedRank rule is applied with the parameters described previously. The precondition of the premise is:

$$\varphi * \lceil R=N\rceil = \lceil x \mapsto R*0\leq R\leq X\rceil * \lceil R=N\rceil$$

Along with the side condition that $N>\ell=0$, we simplify this precondition in Figure 4.4 to $\lceil x \mapsto N*0<N\leq X\rceil$. We next apply the Prob rule in order to analyze the two probabilistic branches in the body of the loop. In the first branch, Assign is used to conclude that $\lceil X \mapsto N-1\rceil$ and in the second branch, Skip is used to conclude that $\lceil x \mapsto N\rceil$. Putting the two branches together, we get that $\lceil x \mapsto N-1\rceil \oplus_{\frac{1}{2}} \lceil x \mapsto N\rceil$, which is roughly equivalent to the following assertion (in Figure 4.4, we omit the bounds on R and N for clarity).

$$\left(\bigoplus_{R\sim\mathbf{unif}(\{N-1\})} \varphi\right) \oplus_{\frac{1}{2}} \left(\bigoplus_{R\sim\mathbf{unif}(\{N\})} \varphi\right)$$

The postcondition is therefore $\varphi[0/R]$, which is equal to $\lceil x \mapsto 0 \rceil$.

4.4.2 The von Neumann Trick

The second running example is a randomness-conserving protocol for simulating a fair coin given a coin with a fixed, unknown bias $0 < p < 1$. Originally due to von Neumann [1951], the protocol (shown as a program below) flips the coin twice and returns the value of the first coin flip if the outcomes differ, otherwise it tries again.

$$
\begin{array}{l}
x \coloneqq 0 \fatsemi \\
y \coloneqq 0 \fatsemi \\
\textbf{while}\ x = y\ \textbf{do} \\
\quad x \coloneqq\approx \textbf{Ber}\,(p) \fatsemi \\
\quad y \coloneqq\approx \textbf{Ber}\,(p)
\end{array}
$$

Intuitively, the von Neumann trick works because on any given round, the probability of getting $x = 0$ and $y = 1$ is equal to the probability of getting $x = 1$ and $y = 0$; both events occur with probability $p(1-p)$. That means that on any given round, the protocol will terminate with probability $2p(1-p)$, and if it does terminate, then the values of x will be uniformly distributed.

To prove correctness of the protocol using pOL, we will need a loop invariant with which we can invoke BoundedRank. We will construct the invariant in two parts, which represent the two different ranks (0 and 1).

$$
\varphi \triangleq (\varphi_0 * \lceil R = 0 \rceil) \vee (\varphi_1 * \lceil R = 1 \rceil) \qquad \varphi_0 \triangleq \bigoplus_{X \sim \textbf{Ber}\left(\frac{1}{2}\right)} \lceil x \mapsto X * y \mapsto \neg X \rceil \qquad \varphi_1 \triangleq \lceil x = y \mapsto 1 \rceil
$$

Rank 0 must imply termination, so $x \neq y$. In addition, for correctness, we require that the values of x are uniformly distributed. In the case where $R = 1$, we only require that the loop guard $x = y$ is true. As stated previously, on every round, the loop will exit with probability $2p(1-p)$, which is nonzero since $0 < p < 1$. The derivation is shown in Figure 4.5.

The precondition is simply φ_1, which states that $x = y$ and obviously applies to the program state after executing $x \coloneqq 0 \fatsemi y \coloneqq 0$. The proof is quite mechanical, except for the application of

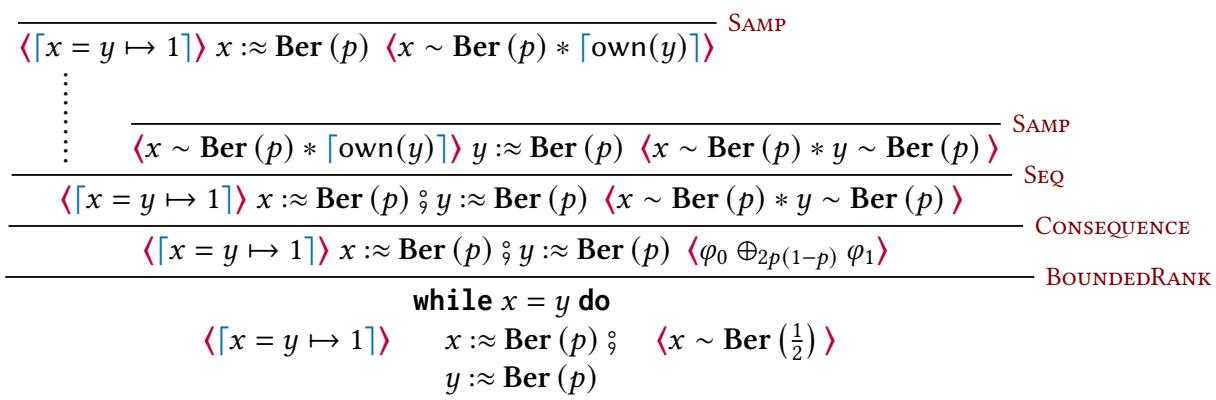


Figure 4.5: Correctness proof for the von Neumann trick in pOL.

the rule of Consequence. To see why the consequence holds, we can first apply the distributive law of the outcome conjunction over the separating conjunction to obtain:

$$x \sim \mathbf{Ber}\,(p) * y \sim \mathbf{Ber}\,(p) \quad \Longrightarrow \quad \bigoplus_{X \sim \mathbf{Ber}(p)} \bigoplus_{Y \sim \mathbf{Ber}(p)} \lceil x \mapsto X * y \mapsto Y \rceil$$

Now, we are going to fold the nested outcome conjunctions into a single Bernoulli distribution by considering only the cases where $X = Y$ and $X \neq Y$. The values of X and Y differ when $X = 0$ and $Y = 1$ or vice versa, which both occur with probability $p(1-p)$, therefore $X \neq Y$ occurs with probability $2p(1-p)$. Furthermore, the probability that $X = 0$ and $X = 1$ are equal, conditioned on the fact that $X \neq Y$, since both probabilities are $p(1-p)$. Therefore, the following implication holds too.

$$\bigoplus_{X \sim \mathbf{Ber}(p)} \bigoplus_{Y \sim \mathbf{Ber}(p)} \lceil x \mapsto X * y \mapsto Y \rceil \quad \Longrightarrow \quad \left(\bigoplus_{X \sim \mathbf{Ber}\left(\frac{1}{2}\right)} \lceil x \mapsto X * y \mapsto \neg X \rceil \right) \oplus_{2p(1-p)} \lceil x = y \mapsto 1 \rceil$$

The assertion above is exactly equivalent to the postcondition of the premise in the application of BoundedRank in Figure 4.5. Accordingly, the final postcondition should be simply φ_0. In Figure 4.5, we simplified φ_0 by forgetting the information about y to obtain simply $x \sim \mathbf{Ber}\left(\frac{1}{2}\right)$.

4.5 RELATED WORK

Weakest Pre-Expectations. Weakest pre-expectation transformers are calculi for reasoning about probabilistic programs in terms of expected values [Morgan et al. 1996a]. They were inspired by propositional weakest precondition calculi [Dijkstra 1975, 1976], Probabilistic Propositional Dynamic Logic [Kozen 1983], and probabilistic predicate transformers [Jones 1990].

The key idea is to define $\mathrm{wp}\colon cmd_{\mathrm{P}} \times exp \to exp$, which takes in a program C and a expression e and returns an expression e', which represents the expected value of e after C is executed. For example, the expected value of $x + y$ after executing $x :\approx \mathbf{unif}\,(\{1,2,3\})$ is derived below:

$$\mathrm{wp}\,(x :\approx \mathbf{unif}\,(\{1,2,3\}), x+y) = \frac{1}{3}\cdot(1+y) + \frac{1}{3}\cdot(2+y) + \frac{1}{3}\cdot(3+y) = 2+y$$

The traditional wp approach can only be used to reason about a single outcome at a time, unlike POL which can reason about many outcomes in tandem. However, reasoning about outcomes and expectations are not mutually exclusive; Barthe et al. [2018, Theorem 1] showed how to embed a wp calculus in a probabilistic program logic. A similar construction could be done in POL. In addition, Zhang et al. [2024] introduced a weakest hyper pre calculus, which can range over multiple outcomes. Refer to Kaminski [2019] for a thorough overview of the wp technique.

Work on wp has intersected with almost sure termination analysis for probabilistic programs. Some of that work uses martingales [Chakarov and Sankaranarayanan 2013] to show that programs terminate with finite expected running time. More sophisticated techniques exist for almost sure termination too, which inspired the rules in this chapter [Kaminski 2019; McIver and Morgan 2005; McIver et al. 2018].

Probabilistic Hoare Logics. Probabilistic Outcome Logic takes inspiration from prior program logics such as Probabilistic Hoare Logic [den Hartog 2002; Corin and den Hartog 2006; den Hartog 1999], VPHL [Rand and Zdancewic 2015], and Ellora [Barthe et al. 2018]. Those logics provide means to prove properties about the distributions of outcomes in probabilistic programs,

to which we added the ability to reason about independence via separation logic, which increases the compositionality and modularity of the proof system.

Couplings and Relational Proofs. An alternative strategy for verifying probabilistic programs is to relate the behavior of two probabilistic programs. Probabilistic Relational Hoare Logic (pRHL) does so by lifting a relation over deterministic states to a relation over distributions of states [Barthe et al. 2009, 2015]. For example, the following specification relates sampling from a Bernoulli distribution to sampling from from a uniform distribution, which is witnessed by a coupling in which a draw of 0 (resp. 1) in the first program corresponds to a draw of 1 (resp. 2) in the second program with equal probability.

$$\{\mathbf{true}\}\ \left(x_1 :\approx \mathbf{Ber}\left(\tfrac{1}{2}\right)\right)\ \sim\ \left(x_2 :\approx \mathbf{unif}\left(\{1,2\}\right)\right)\ \{x_1 = x_2 - 1\}$$

Refer to Hsu [2017] for a comprehensive introduction to probabilistic relational techniques.

Relational logics can be used for refinement proofs, where one program is being verified and the other is the specification [Gregersen et al. 2024a,b; Haselwarter et al. 2024]. It can sometimes be easier to prove a relational property as compared to a unary one, so the refinement approach is useful as long as the specification program is *obviously correct*. However, the downside of the relational approach is that the sampling operations of the two programs have to be coupled in lockstep, and so it is not so easy to prove correctness in cases where the distribution over desired outcomes looks very different from the structure of sampling operations in the program, as is the case in the three-sided die (Section 4.1.2) and the von Neumann trick (Section 7.5.2).

The Clutch logic somewhat mitigates the issue of lockstep execution by introducing a logical pre-sampling tape, which can be manipulated as part of the proof [Gregersen et al. 2024b]. However, that approach introduces additional complexity, whereas POL can often more easily establish a direct specification. Bao et al. [2025] showed how relational probabilistic logics can be combined with Probabilistic Separation Logic to benefit from the features of both approaches.

Probabilistic Separation Logic. The idea of modeling separation using probabilistic independence was first explored by Barthe et al. [2020], however the resulting Probabilistic Separation Logic (PSL) had only limited abilities to reason about the program's control flow. Successors of PSL such as DIBI [Bao et al. 2021], Lilac [Li et al. 2023], and Bluebell [Bao et al. 2025] included additional modalities to reason about conditional independence, which improved handling of control flow. However, none of those logics support the Split1 and Split2 rules of pOL, which are needed to reason about control flow in a more general way.

Lilac supports reasoning about continuous probability, but not mutable state. The Lilac rule for reasoning about control flow (H-If) essentially sequentializes the two branches of the if statements to execute one after the other, which is only valid because all variables are immutable. Bluebell has a splitting rule called c-wp-swap, however it requires ownership over all the program variables, and therefore it is incompatible with the frame rule. By contrast, Split1 and Split2 in Outcome Logic are fully compositional and can be used with Frame. Further, neither Lilac nor Bluebell have unbounded loops, and so they cannot be used to reason about almost surely terminating programs like we saw in Section 4.4.

Amaryllis was more recently introduced to support probabilistic separation in tandem with more sophisticated types of resources such as dynamically allocatable memory, however it requires finitely supported distributions and therefore does not have the dcpo structure that would be needed for unbounded looping [Lohse et al. 2026].

Relative Completeness and Expressivity. The idea of relative completeness was first developed by Cook [1978] as a way to show that a program logic's inference rules are sufficient for reasoning about the structure of a program even if entailments in the underlying assertion language are undecidable. Analogous results for probabilistic programs are more sparse.

The first relative completeness proof for a probabilistic program logic was due to den Hartog [2002], however the programming language did not include loops, and so the metatheory is substantially simplified. Later, Barthe et al. [2018] and Zilberstein [2025] proved relative com-

pleteness for logics with unbounded looping, but their logics are based on semantic assertions and therefore the completeness proofs do not answer questions of what an expressive assertion language would look like. Further, the derivations witnessed by those completeness proofs involve existentially quantifying over the particular distribution that satisfies the precondition, and therefore they do not mirror the style of derivation that would be used in practice.

Batz et al. [2021] developed an expressive assertion language for weakest pre-expectations, but that language does not include the ability to reason about multiple outcomes as pOL does. Majumdar and Sathiyanarayana [2025] showed that the loop rule of McIver et al. [2018] is complete for proving almost sure termination; that result could play a role in a future relative completeness proof for pOL.

CHAPTER 5

DEMONIC OUTCOME LOGIC

Moving on from pOL, we now enter a domain of programs that are not only probabilistic, but also display a second form of nondeterministic behavior, which is controlled by an adversary. Eventually, that adversarial behavior will be used to model a scheduler that chooses how to interleave concurrent computation threads, but as we will see in this chapter, there are major challenges even in a simpler setting where the nondeterminism arises from an explicit choice operator.

5.1 OVERVIEW

We begin by adding a new demonic choice operator to our programming language $C_1 \,\&\, C_2$, and a nondeterministic assignment operator $x \leftarrow e$. Whereas in $C_1 \oplus_p C_2$ and $x :\approx d(e)$, each outcome occurs with a precise probability, the new constructs are fundamentally different We cannot reason about $C_1 \,\&\, C_2$ and $x \leftarrow e$ quantitatively, but rather can only guarantee correctness of a program if the desired property holds even in the *worst case*.

In addition, the new demonic constructs have intricate interactions with probabilistic commands. To demonstrate that interaction, we consider an example in which an adversary tries to guess the outcome of a fair coin flip. If the coin flip happens before the adversary's choice (below on the left), then the adversary can exactly guess the outcome of the coin flip, since its value is already stored in the variable x. However, if the adversary chooses first (below on the right), then x will be uniformly distributed no matter how the adversary picks.

$$x :\approx \mathbf{Ber}\left(\tfrac{1}{2}\right) \fatsemi y \leftarrow \{0,1\} \qquad\qquad y \leftarrow \{0,1\} \fatsemi x :\approx \mathbf{Ber}\left(\tfrac{1}{2}\right)$$

We need a semantic model and sound logical rules to capture the behavior described above. We will now outline the challenges that arise in designing such a logical system.

5.1.1 Semantics and Compositionality

Following the approach to program semantics outlined in Section 2.1, a monad for probabilistic nondeterminism will allow us to easily define semantics for our programs. However, it is well known that the nondeterminism (powerset) monad (Example 2) does not compose with the probability monad (Definition 2.14) in a straightforward way [Varacca and Winskel 2006; Zwart and Marsden 2019; Zwart 2020; Parlant 2020]. As a simple counterexample, let us examine two different parenthesizations of the program below [Varacca and Winskel 2006, §6.5].

$$\left(x :\approx \mathbf{Ber}\left(\tfrac{1}{2}\right) \fatsemi x := 1\right) \fatsemi y \leftarrow \{0,1\} \qquad\qquad x :\approx \mathbf{Ber}\left(\tfrac{1}{2}\right) \fatsemi \left(x := 1 \fatsemi y \leftarrow \{0,1\}\right)$$

Suppose we now wanted to interpret both of the above programs using a naive composition of the powerset and probability monads, where the operations unit $\eta \colon X \to 2^{\mathcal{D}(X)}$ and Kleisli composition $(-)^\dagger \colon (X \to 2^{\mathcal{D}(Y)}) \to 2^{\mathcal{D}(X)} \to 2^{\mathcal{D}(Y)}$ are defined below.

$$\eta(x) = \{\delta_x\} \qquad\qquad f^\dagger(S) = \left\{ \sum_{x \in \mathsf{supp}(\mu)} \mu(x) \cdot \nu_x \;\middle|\; \mu \in S,\ \forall x \in \mathsf{supp}(\mu).\ \nu_x \in f(x) \right\}$$

That is, the unit η simply returns a singleton set containing a point-mass distribution and the Kleisli composition produces a set containing all of the distributions that could result from any combination of demonic choices. It is also easy to see how to define probabilistic and demonic choice operators. A probabilistic choice $S \oplus_p T$ is the set of convex combinations of demonically chosen distributions from S and T, and a demonic choice $S \,\&\, T$ is simply a union of the demonic choices in S and T.

$$S \oplus_p T = \left\{\mu \oplus_p \nu \mid \mu \in S, \nu \in T\right\} \qquad\qquad S \,\&\, T = S \cup T$$

However, the definitions above lead to non-compositional semantics. In the first variant of the program above, the $x := 1$ command completely erases the fact that a probabilistic choice had occurred, and so the subsequent nondeterministic choice happens only once, meaning that y is

either 0 or 1 each with probability 1.

$$
\begin{aligned}
&[\![(x :\approx \mathbf{Ber}\left(\tfrac{1}{2}\right) \fatsemi x := 1) \fatsemi y \leftarrow \{0,1\}]\!] (s) \\
&= [\![y \leftarrow \{0,1\}]\!]^\dagger ([\![x :\approx \mathbf{Ber}\left(\tfrac{1}{2}\right) \fatsemi x := 1]\!] (s)) \\
&= [\![y \leftarrow \{0,1\}]\!] (s[x := 1]) \\
&= \left\{ \left[s[x := 1, y := 0] \mapsto 1 \right], \left[s[x := 1, y := 1] \mapsto 1 \right] \right\}
\end{aligned}
$$

However, in the second variant, the demonic choice is grouped with $x := 1$, and the entire composite command is executed once for each of the two outcomes of the Bernoulli distribution. That means that in the second program, the demonic choice can be resolved differently for each outcome of the initial sampling operation, so there is an additional possibility that y is 0 or 1 each with probability $\frac{1}{2}$.

$$
\begin{aligned}
&[\![x :\approx \mathbf{Ber}\left(\tfrac{1}{2}\right) \fatsemi (x := 1 \fatsemi y \leftarrow \{0,1\})]\!] (s) \\
&= [\![x := 1 \fatsemi y \leftarrow \{0,1\}]\!]^\dagger ([\![x :\approx \mathbf{Ber}\left(\tfrac{1}{2}\right)]\!] (s)) \\
&= [\![x := 1 \fatsemi y \leftarrow \{0,1\}]\!]^\dagger \left(\left\{ \begin{bmatrix} s[x := 0] & \mapsto & 1/2 \\ s[x := 1] & \mapsto & 1/2 \end{bmatrix} \right\} \right) \\
&= \left\{ \left[s[x := 1, y := 0] \mapsto 1 \right], \left[s[x := 1, y := 1] \mapsto 1 \right], \begin{bmatrix} s[x := 1, y := 0] & \mapsto & 1/2 \\ s[x := 1, y := 1] & \mapsto & 1/2 \end{bmatrix} \right\}
\end{aligned}
$$

Following from the counterexample above, sets of distributions do not constitute a monad since the associative law $(f^\dagger \circ g)^\dagger = f^\dagger \circ g^\dagger$ is violated. More concerningly, associativity of sequential composition $C_1 \fatsemi (C_2 \fatsemi C_3) \equiv (C_1 \fatsemi C_2) \fatsemi C_3$ does not hold and the Seq rule of pOL (and other Hoare-like logics) is unsound, meaning that sequential programs cannot be analyzed compositionally.

However, compositionality can be restored by requiring the sets of distributions used to encode program semantics to be *convex*. A set $S \in 2^{\mathcal{D}(X)}$ is convex if for any $\mu, \nu \in S$ and $p \in [0,1]$ the convex combination $\mu \oplus_p \nu$ is also contained in S. In restricting to convex sets, it is no longer valid to define demonic choice as a simple union, since convex sets are not closed under union. Instead, demonic choice becomes a convex union.

$$
S \,\&\, T = \bigcup_{p \in [0,1]} S \oplus_p T
$$

Operationally, the convex union definition of demonic choice corresponds to an adversary that can use coin flips to make decisions. Instead of deterministically picking between two options, the adversary can flip a coin with any bias p to make the choice. Of course, the adversary could always choose $p = 0$ or $p = 1$ to make the choice deterministic, so the probabilistic adversary is strictly a stronger threat model. Convexity solves the associativity program, since in both variants of the previous program, y can be 0 or 1 with any probability. That is, we get:

$$\begin{aligned}\left[\!\left[\left(x :\approx \mathbf{Ber}\left(\tfrac{1}{2}\right) \fatsemi x := 1\right) \fatsemi y \leftarrow \{0,1\}\right]\!\right](s) &= \left[\!\left[x :\approx \mathbf{Ber}\left(\tfrac{1}{2}\right) \fatsemi \left(x := 1 \fatsemi y \leftarrow \{0,1\}\right)\right]\!\right](s) \\ &= \left\{ \begin{bmatrix} s[x := 1, y := 0] & \mapsto & p \\ s[x := 1, y := 1] & \mapsto & 1-p \end{bmatrix} \;\middle|\; p \in [0,1] \right\}\end{aligned}$$

5.1.2 Demonic Outcome Logic

In Demonic Outcome Logic (DOL), we specify the probabilistic behavior of programs using the same outcome assertions that we used in POL. However, the demonic triple $\langle\varphi\rangle\ C\ \langle\psi\rangle$ has a subtly different meaning; since the denotation of C is now a *set* of distributions, ψ must model every distribution in that set. The handling of nondeterminism by universal quantification over the final distributions mirrors the way that nondeterminism is captured in standard Hoare Logic.

However, unlike Hoare Logic, the pre- and postconditions in DOL describe probabilistic behavior. So, disjunctions are not expressive enough for reasoning about nondeterminism. For instance, $\lceil y \mapsto 0\rceil \vee \lceil y \mapsto 1\rceil$ is not a valid postcondition for the program $y \leftarrow \{0,1\}$; it is not true that either $y \mapsto 0$ with probability 1 or $y \mapsto 1$ with probability 1, but rather rather the values of y may be distributed in some other way.

To account for the probabilistic scheduling behavior, we introduce a new $\binampersand$ logical modality, which is a nondeterministic analogue of the outcome conjunction. The assertion $\binampersand_{X \in S}\, \varphi$ means that there exists some distribution $\mu \in \mathcal{D}(S)$ such that $\bigoplus_{X \sim \mu} \varphi$ holds. We can use $\binampersand$ to specify the behavior of a demonic choice.

$$\langle\lceil y \mapsto -\rceil\rangle\ y \leftarrow \{0,1\}\ \langle \binampersand_{Y \in \{0,1\}} \lceil y \mapsto Y\rceil\rangle$$

Since $\&$ is formulated with the same measure theoretic foundations as the outcome conjunction, it can also be used to reason about independence, even if we do not know the precise probabilities of each outcome. Returning to the variant of the coin flip game where the adversary picks first $y \leftarrow \{0,1\} \fatsemi x :\approx \mathbf{Ber}\left(\frac{1}{2}\right)$, we can use the FRAME rule on the second command to conclude that the coin flip is probabilistically independent from y, a previously chosen demonic value.

$$\frac{\langle \lceil x \mapsto - \rceil \rangle\ x :\approx \mathbf{Ber}\left(\frac{1}{2}\right)\ \langle x \sim \mathbf{Ber}\left(\frac{1}{2}\right) \rangle}{\langle \lceil x \mapsto - \rceil * \&_{Y \in \{0,1\}} \lceil y \mapsto Y \rceil \rangle\ x :\approx \mathbf{Ber}\left(\frac{1}{2}\right)\ \langle \left(x \sim \mathbf{Ber}\left(\frac{1}{2}\right)\right) * \&_{Y \in \{0,1\}} \lceil y \mapsto Y \rceil \rangle}\ \textsc{Frame}$$

Although we do not know the exact distribution over Y, the probabilistic independence property means that we can apply a distributive law to obtain the following:

$$\begin{aligned}\left(x \sim \mathbf{Ber}\left(\tfrac{1}{2}\right)\right) * \mathop{\&}_{Y \in \{0,1\}} \lceil y \mapsto Y \rceil &\implies \mathop{\&}_{Y \in \{0,1\}} \left(x \sim \mathbf{Ber}\left(\tfrac{1}{2}\right)\right) * \lceil y \mapsto Y \rceil \\ &\implies \mathop{\&}_{Y \in \{0,1\}} \mathop{\oplus}_{X \sim \mathbf{Ber}\left(\frac{1}{2}\right)} \lceil x \mapsto X * y \mapsto Y \rceil\end{aligned}$$

So, for any fixed Y, X is uniformly distributed. That means that $x = y$ with probability exactly $\frac{1}{2}$.

$$\implies \mathop{\&}_{Y \in \{0,1\}} \mathop{\oplus}_{X \sim \mathbf{Ber}\left(\frac{1}{2}\right)} \lceil (x = y) \mapsto X \rceil$$

Since the inner term does not depend on Y, we can remove the $\&$ to determine that $x = y$ with probability $\frac{1}{2}$, *i.e.,* that the adversary cannot do any better than guessing the value of x.

$$\implies (x = y) \sim \mathbf{Ber}\left(\tfrac{1}{2}\right)$$

Modular reasoning with the FRAME rule gives us an elegant way to specify the program above. However the reversed program, where the flip occurs before the demonic choice, is more complex.

5.1.3 The Pitfalls of Reasoning about Nondeterminism

When reasoning about nondeterminism in DOL, it is important to take into account the ways in which the demonic adversary can inject correlations between the program variables. Let us return to the previous specification for a demonic choice.

$$\langle \lceil y \mapsto - \rceil \rangle\ y \leftarrow \{0,1\}\ \langle \&_{Y \in \{0,1\}} \lceil y \mapsto Y \rceil \rangle$$

The triple above is valid in isolation, but it is not *frame preserving*. Indeed, applying the Frame rule with $x \sim \mathbf{Ber}\left(\frac{1}{2}\right)$ would allow us to conclude that the demonically chosen y is probabilistically independent from a previously randomly sampled variable x, which is not true according to the program semantics since y can be chosen based on the value of x.

So, reasoning about nondeterminism requires a weaker triple. Although we just saw that the weaker triple cannot support a Frame rule with independence guarantees, it can support a Frame rule that only guarantees disjointness of state. To achieve that, we generalize the separating conjunction $\varphi *_m \psi$ to be parameterized by a mode $m \in \{\mathsf{w}, \mathsf{s}\}$, where w stands for weak separation, which does not imply independence, and s stands for strong separation and has the same meaning that we saw in Chapter 3. Strong is the default mode, so we will henceforth write $*$ to mean $*_\mathsf{s}$. Triples are accordingly also parameterized by a mode, which indicates what kind of separating conjunction can be used in the frame rule.

$$\frac{\vdash_m \langle \varphi \rangle\ C\ \langle \psi \rangle}{\vdash_m \langle \varphi *_m \vartheta \rangle\ C\ \langle \psi *_m \vartheta \rangle}\ \textsc{Frame}$$

Although most rules are agnostic to the kind of frame preservation that is enabled, certain treatments of nondeterminism require a weak triple, such as handling demonic choice.

$$\vdash_\mathsf{w} \langle \lceil y \mapsto - \rceil \rangle\ y \leftarrow \{0,1\}\ \langle\ \&_{Y \in \{0,1\}} \lceil y \mapsto Y \rceil \rangle$$

Now, we can use the weak variant of the Frame rule to get a specification for a random choice followed by a demonic choice

$$\frac{\vdash_\mathsf{w} \langle \lceil y \mapsto - \rceil \rangle\ y \leftarrow \{0,1\}\ \langle\ \&_{Y \in \{0,1\}} \lceil y \mapsto Y \rceil \rangle}{\vdash_\mathsf{w} \langle\ \left(x \sim \mathbf{Ber}\left(\tfrac{1}{2}\right)\right) *_\mathsf{w} \lceil y \mapsto - \rceil \rangle\ y \leftarrow \{0,1\}\ \langle\ \left(x \sim \mathbf{Ber}\left(\tfrac{1}{2}\right)\right) *_\mathsf{w} \&_{Y \in \{0,1\}} \lceil y \mapsto Y \rceil \rangle}\ \textsc{Frame}$$

Since x and y are stored in separate portions of memory, the demonic assignment to y cannot affect the way that x is distributed. However, the scheduler can base its choice of y on the distribution of values over x, so x and y are not necessarily probabilistically independence. That intuition is exactly what is formally conveyed by the triple above.

While it may seem inconvenient to have two universes of triples, the strong and weak triples are compatible with each other, and there are rules to go between them. Unsurprisingly, any strong triple can be converted into a weak triple. However, weak triples can also be strengthened into strong triples as long as the postcondition is *precise*.

$$\frac{\vdash \langle \varphi \rangle\ C\ \langle \psi \rangle}{\vdash_{\mathrm{w}} \langle \varphi \rangle\ C\ \langle \psi \rangle}\textsc{Weaken} \qquad \frac{\vdash_{\mathrm{w}} \langle \varphi \rangle\ C\ \langle \psi \rangle \qquad \mathsf{precise}(\psi)}{\vdash \langle \varphi \rangle\ C\ \langle \psi \rangle}\textsc{Strengthen}$$

Recall from Definition 3.10 that a precise assertion is one where every measurable event has an exact probability. Obviously, $\&_{X \in \{0,1\}} \lceil y \mapsto Y \rceil$ is not precise, since the probabilities of y being 0 or 1 are existentially quantified. Precision effectively removes the nondeterminism from the postcondition ψ, ensuring that it is probabilistically independent from the frame by limiting the behavior of the adversary to one unique model. Some amount of nondeterminism may be possible, but it is not measurable.

In the coin flip game example, we can recover a strong triple by weakening the postcondition from $\&_{X \in \{0,1\}} \lceil y \mapsto Y \rceil$ to $\lceil y \in \{0,1\} \rceil$, which is precise since it is valid in any probability space where $\lceil y \mapsto 1 \rceil$ and $\lceil y \mapsto 0 \rceil$ are not measurable, but $\lceil y \in \{0,1\} \rceil$ has probability exactly 1.

$$\dfrac{\dfrac{\dfrac{\vdash_{\mathrm{w}} \langle \lceil y \mapsto - \rceil \rangle\ y \leftarrow \{0,1\}\ \langle \&_{Y \in \{0,1\}} \lceil y \mapsto Y \rceil \rangle}{\vdash_{\mathrm{w}} \langle \lceil y \mapsto - \rceil \rangle\ y \leftarrow \{0,1\}\ \langle \lceil y \in \{0,1\} \rceil \rangle}\textsc{Consequence}}{\vdash \langle \lceil y \mapsto - \rceil \rangle\ y \leftarrow \{0,1\}\ \langle \lceil y \in \{0,1\} \rceil \rangle}\textsc{Stengthen}}{\vdash \left\langle \left(x \sim \mathbf{Ber}\left(\tfrac{1}{2}\right)\right) * \lceil y \mapsto - \rceil \right\rangle\ y \leftarrow \{0,1\}\ \left\langle \left(x \sim \mathbf{Ber}\left(\tfrac{1}{2}\right)\right) * \lceil y \in \{0,1\} \rceil \right\rangle}\textsc{Frame}$$

Note that the specification above does not imply that x and y are probabilistically independent; since we only have almost-surely-true information about y, it is possible that it is correlated with x in the concrete distribution of outcomes. Therefore, we cannot conclude anything about the probability that $x = y$, unlike in the case where the adversary picks first.

5.2 DENOTATIONAL SEMANTICS AND THE CONVEX POWERDOMAIN

The addition of demonic choice to our logical system requires us to first adjust the syntax and semantics of the underlying programming language. Figure 5.1 gives the syntax of the Probabilis-

$$
\begin{array}{rll}
cmd_{\text{PD}} \ni C ::= & \textbf{skip} & \textit{(No-op)} \\
& \mid C_1 \fatsemi C_2 & \textit{(Sequential Composition)} \\
& \mid C_1 \oplus_p C_2 & \textit{(Probabilistic Choice)} \\
& \mid C_1 \,\&\, C_2 & \textit{(Demonic Choice)} \\
& \mid \textbf{if}\ b\ \textbf{then}\ C_1\ \textbf{else}\ C_2 & \textit{(Conditional Branching)} \\
& \mid \textbf{while}\ b\ \textbf{do}\ C & \textit{(Loops)} \\
& \mid a & \textit{(Basic Actions } a \in act\textit{)} \\
\\
act \ni a ::= & x := e & \textit{(Assignment)} \\
& \mid x :\approx d(e) & \textit{(Probabilistic Sampling)} \\
& \mid x \leftarrow e & \textit{(Demonic Assignment)}
\end{array}
$$

Figure 5.1: Syntax of the Probabilistic Demonic Command Language (pdCL). The new additions over pCL are highlighted in yellow. As before, tests $b \in test$ and expressions $e \in exp$ are discussed in Section 2.1.1 and distributions d are from Figure 2.3

tic Demonic Command Language (pdCL), where the demonic choice commands from Section 5.1 have been added. We will define denotational semantics for pdCL using a computational domain with a monad and dcpo structure, as in Section 2.1. That domain is fundamentally based on sets of distributions $2^{\mathcal{D}(\mathsf{Mem}[S]_\perp)}$, but as we already saw in Section 5.1.1, simple sets of distributions are not a monad on their own [Varacca and Winskel 2006], and therefore do not provide the compositional properties that are needed to support the metatheory of a program logic.

5.2.1 The Convex Powerdomain

We now present the convex powerdomain, the underlying semantic structure that we will use to represent probabilistic nondeterminism. We use a construction that is due to He et al. [1997] and McIver and Morgan [2005]. To recover the monad structure, sets of distributions must be convex, which intuitively means that they are closed under convex combinations of their elements.

Definition 5.1 (Convex Sets). *A set $S \subseteq \mathcal{D}(X)$ of distributions is* convex *if $\mu \in S$ and $\nu \in S$ implies that $\mu \oplus_p \nu \in S$ for every $p \in [0,1]$.*

Additional restrictions are needed for the DCPO structure, so that loops have well defined semantics. Sets of distributions are ordered using the Smyth [1978] order.

$$S \sqsubseteq_{\mathsf{Smyth}} T \qquad \text{iff} \qquad \forall \nu \in T.\quad \exists \mu \in S.\quad \mu \sqsubseteq_{\mathcal{D}} \nu$$

We use the Smyth order because is models total correctness [Søndergaard and Sestoft 1992], and therefore it will provide a foundation for establishing almost sure termination, as we did in POL. The problem is that the Smyth order is not a partial order, since it is not antisymmetric. To make the Smyth order antisymmetric, we must additionally require that the sets are up-closed.

Definition 5.2 (Up-Closed and Down-Closed Sets)**.** *Given a poset $\langle A, \leq \rangle$, a set $S \subseteq A$ is up-closed (resp. down-closed) if $x \in S$ and $x \leq y$ implies that $y \in S$ (resp. $x \in S$ and $y \leq x$ implies that $y \in S$). The up- and down-closures of an element $x \in A$ are defined as follows:*

$$\uparrow x \triangleq \{y \in A \mid x \leq y\} \qquad\qquad \downarrow x \triangleq \{y \in A \mid y \leq x\}$$

As a convention if $\langle A, < \rangle$ is a strict poset, then $\uparrow x$ and $\downarrow x$ do not contain x. Closures extend to sets of elements $X \subseteq A$ as follows:

$$\uparrow X \triangleq \bigcup_{x \in X} \uparrow x \qquad\qquad \downarrow X \triangleq \bigcup_{x \in X} \downarrow x$$

Thus, X is up-closed (resp. down-closed) iff $X = \uparrow X$ (resp. $X = \downarrow X$).

Recall from Lemma 2.11 that $\mu \sqsubseteq_{\mathcal{D}} \nu$ implies that $\nu(\bot) \leq \mu(\bot)$, and so if $\mu(\bot) = 0$, then μ is already maximal. Therefore if $\mu(\bot) = 0$, then $\uparrow\{\mu\} = \{\mu\}$, but if $\mu(\bot) > 0$, then μ can be made larger by reassigning probability mass from $\bot$ to proper states, *e.g.*, $\delta_\bot \sqsubseteq_{\mathcal{D}} \delta_\sigma$. As a consequence, $\uparrow\{\delta_\bot\} = \mathcal{D}(\mathsf{Mem}[V]_\bot)$, the set of all distributions. Up-closure means that we cannot be sure whether a program truly diverges, or instead exhibits erratic nondeterministic behavior, which is a common limitation of the Smyth powerdomain [Søndergaard and Sestoft 1992]. However, DOL is concerned with proving almost sure termination of programs, so the loss of precision in the semantics when nontermination might occur does not affect the accuracy of our logic.

When restricting to up-closed sets, the Smyth order is equivalent to reverse subset inclusion, *i.e.*, $S \sqsubseteq_{\mathsf{Smyth}} T$ iff $S \supseteq T$. That means that suprema are given by set intersection. Although intersections of arbitrary sets of sets are always defined, we are interested only in nonempty sets. Since nontermination is captured by $\uparrow\{\delta_\perp\} = \mathcal{D}(\mathsf{Mem}[V]_\perp)$, an empty set of distributions has no inherent meaning as a semantic object. Therefore, we need a way to guarantee that intersections of directed sets are nonempty.

To guarantee nonemptiness, we add a final requirement that the sets of distributions must be Cauchy closed, meaning that they are closed in the usual topological sense [McIver and Morgan 2005]. A subset of $\mathcal{D}(X)$ is *closed* if it is closed in the product topology $[0,1]^X$, where $[0,1]$ has the Euclidean topology. So closure means that a set S contains all of its limit points.

As a consequence of Cauchy closure, the convex powerdomain cannot model unbounded nondeterminism. More precisely, it is impossible to define a primitive command $x := \bigstar$, which terminates and nondeterministically selects a value for x from an infinite set (such as $\mathbb{N}$). Bounded nondeterminism is typical a limitation, stemming from the impossibility proof of Apt and Plotkin [1986]. Refer to McIver and Morgan [2005, p.335] for a more explicit discussion of how Cauchy closure relates to bounded nondeterminism. We now have all the ingredients needed to define the convex powerdomain:

Definition 5.3 (The Convex Powerdomain)**.**

$$C(X) \triangleq \left\{S \subseteq \mathcal{D}(X_\perp) \mid S \text{ is nonempty, convex, Cauchy closed, and up-closed}\right\}$$

As described previously, the various restrictions on C give it the required properties to serve as a semantic domain, as outlined in Section 2.1.5. Convex powersets have a monad structure, originally due to Jacobs [2008].

Lemma 5.4. *The structure* $\left\langle C, \eta, (-)^\dagger \right\rangle$ *is a monad, where the operations are defined as follows:*

$$\eta(x) \triangleq \uparrow\{\delta_x\} \qquad f^\dagger(S) \triangleq \left\{ \sum_{x \in \mathsf{supp}(\mu)} \mu(x) \cdot \nu_x \;\middle|\; \mu \in S, \forall x \in \mathsf{supp}(\mu).\ \nu_x \in f_\perp(x) \right\}$$

Proof. See Appendices B.1 and B.2 of Zilberstein et al. [2025b]. □

As stated previously, elements in C are ordered with the Smyth [1978] order, which is equivalent to reverse subset inclusion since elements of C are up-closed. We now prove that the convex powerdomain is a DCPO.

Theorem 5.5. *For any X, $\langle C(X), \sqsubseteq_C \rangle$ is a pointed DCPO, where:*

$$S \sqsubseteq_C T \qquad \textit{iff} \qquad S \sqsubseteq_{\mathsf{Smyth}} T \qquad \textit{iff} \qquad S \supseteq T$$

Proof. The poset structure is immediate from the fact that $\left\langle 2^Y, \supseteq \right\rangle$ is a poset for any Y. We now show that any directed set has a supremum in $C(X)$. By Corollary 2 of Markowsky [1976], it suffices to show that any chain in $C(X)$ has a supremum. Consider such a chain:

$$S_0 \supseteq S_1 \supseteq S_2 \supseteq \cdots$$

We need to show that $\bigcap_i S_i \in C(X)$. By Lemma B.4.4 of McIver and Morgan [2005], $\mathcal{D}(X_\perp)$ is a compact topological space and by the definition of C, each S_i is closed and nonempty. Therefore, by the finite intersection property, $\bigcap_i S_i$ is nonempty. It is well known that the remaining properties (up-closure, convexity, and closedness) are preserved by set intersection.

The bottom of $\langle C(X), \sqsubseteq_C \rangle$ is the set of all distributions $\perp_C = \mathcal{D}(X_\perp)$, which is trivially a superset of any other element of $C(X)$, therefore $C(X)$ is pointed. □

Zilberstein et al. [2025b, Appendix B.3] showed that the Kleisli composition is Scott continuous in $\langle C(X), \sqsubseteq_C \rangle$, therefore C satisfies all the properties needed to give semantics to programs according to the approach in Section 2.1.5.

5.2.2 Denotational Semantics

The convex powerdomain semantics for PDCL is shown in Figure 5.2. Like PCL, the semantics is based on the standard model from Figure 2.2 for all the deterministic commands.

The semantics of the probabilistic choice $C_1 \oplus_e C_2$ and nondeterministic choice $C_1 \,\&\, C_2$ commands rely on new operators on the convex powerdomain. A probabilistic choice $S \oplus_p T$ between

$$
\begin{aligned}
[\![\mathbf{skip}]\!]_C(\sigma) &\triangleq \eta(\sigma) \\
[\![C_1 \fatsemi C_2]\!]_C(\sigma) &\triangleq [\![C_2]\!]_C^\dagger([\![C_1]\!]_C(\sigma)) \\
[\![C_1 \oplus_e C_2]\!]_C(\sigma) &\triangleq [\![C_1]\!]_C(\sigma) \oplus_{[\![e]\!]_{exp}(\sigma)} [\![C_2]\!]_C(\sigma) \\
[\![C_1 \,\&\, C_2]\!]_C(\sigma) &\triangleq [\![C_1]\!]_C(\sigma) \,\&\, [\![C_2]\!]_C(\sigma) \\
[\![\mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2]\!]_{\mathcal{D}}(\sigma) &\triangleq \begin{cases} [\![C_1]\!]_C(\sigma) & \text{if } [\![b]\!]_{test}(\sigma) = 1 \\ [\![C_2]\!]_C(\sigma) & \text{if } [\![b]\!]_{test}(\sigma) = 0 \end{cases} \\
[\![\mathbf{while}\ b\ \mathbf{do}\ C]\!]_C(\sigma) &\triangleq \mathsf{lfp}\left(\Phi_{\langle C,b\rangle}\right)(\sigma) \\
[\![x := e]\!]_C(\sigma) &\triangleq \eta\left(\sigma[x := [\![e]\!]_{exp}(\sigma)]\right) \\
[\![x :\approx d(e)]\!]_C(\sigma) &\triangleq \left\{ \sum_{v \in \mathsf{Val}} d\left([\![e]\!]_{exp}(\sigma)\right)(v) \cdot \delta_{\sigma[x:=v]} \right\} \\
[\![x \leftarrow e]\!]_C(\sigma) &\triangleq \bigAnd_{v \in [\![e]\!]_{exp}(\sigma)} \eta(\sigma[x := v])
\end{aligned}
$$

Figure 5.2: Denotational semantics $[\![-]\!]_C : cmd_{\text{PD}} \to \mathsf{Mem}[V] \to C(\mathsf{Mem}[V])$ of PDCL. The semantics of standard commands carry over from Figure 2.2.

S and T is the set of all distributions obtained by a convex combination $\mu \oplus_p \nu$ for any $\mu \in S$ and $\nu \in T$. Similarly, a nondeterministic choice $S \,\&\, T$ is the union of all probabilistic choices $S \oplus_p T$ for all $p \in [0,1]$, which corresponds to the intuition that the scheduler is probabilistic.

$$
S \oplus_p T \triangleq \{\mu \oplus_p \nu \mid \mu \in S, \nu \in T\} \qquad\qquad S \,\&\, T \triangleq \bigcup_{p \in [0,1]} S \oplus_p T
$$

Indexed nondeterministic choices $\bigAnd_{i \in I} S_i$ are also defined. If the index set is finite, *i.e.*, there is an $n \in \mathbb{N}$ such that $I = \{i_1, \ldots, i_n\}$, then the nondeterministic choice is simply given by iterating binary choices $\bigAnd_{i \in I} S_i \triangleq S_{i_1} \,\&\, \cdots \,\&\, S_{i_n}$. If I is infinite, then $\bigAnd_{i \in I} S_i \triangleq \bot_C$. Refer to McIver and Morgan [2005, p.336] for more information on why the convex closure of an infinite union must be equal to $\bot_C$. For integer limits ℓ and h, we also write $\bigAnd_{i=\ell}^{h} S_i$ to mean $\bigAnd_{i \in \{\ell,\ldots,h\}} S_i$.

The semantics of sampling commands $x :\approx d(e)$ is a singleton set containing a distribution where x is updated to the various possible values of $d(e)$. The semantics of nondeterministic assignment $x \leftarrow e$ is given by an indexed nondeterministic choice.

Although the semantics of the probabilistic commands $C_1 \oplus_e C_2$ and $x :\approx d(e)$ are defined slightly differently in pdCL compared to pCL (Figure 2.4), the intuitive meaning is effectively the same. We show that pdCL is an extension of pCL with the following theorem, which states that the pdCL semantics of a program C is the upward closure of its pCL semantics.

Theorem 5.6 (pdCL and pCL semantics)**.** *For any program $C \in cmd_p$ and memory $\sigma \in \mathsf{Mem}[V]$:*

$$[\![C]\!]_C(\sigma) = \uparrow [\![C]\!]_{\mathcal{D}}(\sigma)$$

Proof. By induction on the structure of C. The cases for **skip**, $C_1 \oplus_e C_2$, **if** b **then** C_1 **else** C_2, $x := e$, and $x :\approx d(e)$ follow from the semantic definitions and the induction hypothesis. For $C_1 \fatsemi C_2$, we additionally need that $\uparrow f^\dagger(\mu) = (\uparrow f)^\dagger(\uparrow \mu)$ for all $f : \mathsf{Mem}[V] \to \mathcal{D}(\mathsf{Mem}[V])$ and $\mu \in \mathcal{D}(\mathsf{Mem}[V])$, where $(\uparrow f)(x) = \uparrow(f(x))$. Expanding the definitions, we must show that:

$$\uparrow\left(\sum_{\sigma \in \mathsf{supp}(\mu)} \mu(\sigma) \cdot f_\perp(\sigma)\right) = \left\{ \sum_{\sigma \in \mathsf{supp}(\nu)} \nu(\sigma) \cdot \xi_\sigma \;\middle|\; \nu \in \uparrow\mu,\ \forall \sigma.\ \xi_\sigma \in \uparrow f_\perp(\sigma) \right\}$$

That equality clearly holds since the Kleisli composition on the right produces an element of $C(\mathsf{Mem}[V])$, which is up-closed by definition, and therefore both the left and right hand side are up-closed sets generated from the single element $\sum_{\sigma \in \mathsf{supp}(\mu)} \mu(\sigma) \cdot f_\perp(\sigma)$.

For while loops, we need to show that up-closure commutes with supremum, *i.e.,* $\uparrow \sup D = \bigcap_{\mu \in D} \uparrow \mu$ for any directed set $D \subseteq \mathcal{D}(\mathsf{Mem}[V])$. We show the inclusion in both directions. First, suppose that $\nu \in \uparrow \sup D$, which means that $\sup D \sqsubseteq_{\mathcal{D}} \nu$. Since $\sup D$ is an upper bound of D, then $\mu \sqsubseteq_{\mathcal{D}} \sup D \sqsubseteq_{\mathcal{D}} \nu$ for all $\mu \in D$, and therefore $\nu \in \uparrow \mu$ for all $\mu \in D$, so $\nu \in \bigcap_{\mu \in D} \uparrow \mu$. For the reverse inclusion, suppose that $\nu \in \bigcap_{\mu \in D} \uparrow \mu$, which means that $\mu \sqsubseteq_{\mathcal{D}} \nu$ for all $\mu \in D$. So, ν is an upper bound of D, which means it must be greater than the least upper bound: $\sup D \sqsubseteq_{\mathcal{D}} \nu$, which means that $\nu \in \uparrow \sup D$. □

Recall that the upward closure operation does not add anything to the semantics of terminating programs where $[\![C]\!]_{\mathcal{D}}(\sigma)(\perp) = 0$. Therefore, if C almost surely terminates starting from state σ, we can simplify Theorem 5.6 to simply say that the pdCL semantics is a singleton set containing the pCL semantics $[\![C]\!]_C(\sigma) = \{[\![C]\!]_{\mathcal{D}}(\sigma)\}$.

$$
\begin{array}{lll}
\varphi ::= \top & & \textit{(True)} \\
\quad \mid \bot & & \textit{(False)} \\
\quad \mid \varphi \wedge \psi & & \textit{(Conjunction)} \\
\quad \mid \varphi \vee \psi & & \textit{(Disjunction)} \\
\quad \mid \exists X.\, \varphi & & \textit{(Existential Quantification)} \\
\quad \mid \varphi *_m \psi & & \textit{(Separating Conjunction)} \\
\quad \mid \bigoplus_{X \sim d(E)} \varphi & & \textit{(Outcome Conjunction)} \\
\quad \mid \mathop{\&}_{X \in E} \varphi & & \textit{(Demonic Outcome Conjunction)} \\
\quad \mid \lceil P \rceil & & \textit{(Almost Sure Assertion)}
\end{array}
$$

Figure 5.3: Outcome Assertion syntax extended with demonic constructs, where $m \in \{\mathsf{s}, \mathsf{w}\}$.

5.3 DEMONIC OUTCOME ASSERTIONS

In order to reason explicitly about nondeterminism, we need to augment the outcome assertions that were first introduced in Chapter 3. The new additions to the outcome assertion syntax are shown in Figure 5.3. The first new construct is the weak separating conjunction $\varphi *_{\mathsf{w}} \psi$, which requires that the variables referenced in φ and ψ are disjoint, but allows probabilistic correlation between them. The second addition is the demonic outcome conjunction $\&_{X \in E}\, \varphi$, where E indexes the *support* of the distribution of outcomes, but does not specify the likelihood of each outcome.

Demonic outcome assertions are modeled similarly to standard outcome assertions. An assertion φ is satisfied by a pair of a logical variable context $\Gamma : \mathsf{LVar} \to \mathsf{Val}$ and probability space $\mathcal{P}$, *i.e.,* $\Gamma, \mathcal{P} \vDash \varphi$. In fact, the definition of the satisfaction relation $\vDash$ is carries over from Figure 3.2 for all but the two new syntactic constructs. In the following sections, we will describe the semantics of the two new assertions, and provide new entailment laws for reasoning about them.

5.3.1 Weak Separation

Rather than defining weak separation as a new logical operator, we generalize the separating conjunction to be parameterized by a mode $m \in \{\mathsf{s}, \mathsf{w}\}$. Strong separation (mode s) is the interpretation of separation from Chapter 3 that requires *probabilistic independence* and separation of variables, whereas weak separation (mode w) does not require independence, and only separates the variables. As we explained in Section 5.1, weak separation is needed when reasoning about nondeterministic outcomes leaves us in a scenario where strong frame preservation does not hold. Semantically, the difference is captured by the combinator operation $\diamond_m$, with strong separation using an independent product and weak separation including all probability spaces $\mathcal{P}$, which marginalize to the two operands.

$$\mathcal{P}_1 \diamond_{\mathsf{s}} \mathcal{P}_2 \triangleq \{\mathcal{P}_1 \otimes \mathcal{P}_2\} \qquad \mathcal{P}_1 \diamond_{\mathsf{w}} \mathcal{P}_2 \triangleq \{\mathcal{P} \mid \mathcal{P}_1 = \pi_U(\mathcal{P}), \mathcal{P}_2 = \pi_V(\mathcal{P})\}$$

where $\Omega_{\mathcal{P}_1} = \mathsf{Mem}[U]$ and $\Omega_{\mathcal{P}_2} = \mathsf{Mem}[V]$ and $U \cap V = \emptyset$. Using the new $\diamond_m$ operator, we can define semantics for the generalized separating conjunction.

$$\Gamma, \mathcal{P} \vDash \varphi *_m \psi \qquad \text{iff} \qquad \begin{array}{l} \mathcal{P}' \preceq \mathcal{P} \ \text{ and } \ \Gamma, \mathcal{P}_1 \vDash \varphi \ \text{ and } \ \Gamma, \mathcal{P}_2 \vDash \psi \\ \text{for some } \ \mathcal{P}_1, \mathcal{P}_2, \text{and } \mathcal{P}' \in \mathcal{P}_1 \diamond_m \mathcal{P}_2 \end{array}$$

Since $\mathcal{P}_1 \diamond_{\mathsf{s}} \mathcal{P}_2$ is the singleton set $\{\mathcal{P}_1 \otimes \mathcal{P}_2\}$, $\varphi *_{\mathsf{s}} \psi$ has the exact same meaning as $\varphi * \psi$ from Chapter 3. Strong separation will be used more commonly, so we will drop the subscript there and write $*$ to mean $*_{\mathsf{s}}$. In addition, $\mathcal{P}_1 \diamond_{\mathsf{s}} \mathcal{P}_2 \subseteq \mathcal{P}_1 \diamond_{\mathsf{w}} \mathcal{P}_2$, which immediately gives us that $\varphi *_{\mathsf{s}} \psi \Rightarrow \varphi *_{\mathsf{w}} \psi$, motivating the *strong* vs. *weak* nomenclature.

5.3.2 Nondeterministic Outcome Conjunctions

The semantics of the nondeterministic outcome conjunction $\&$ is defined by existentially quantifying the distribution in the definition of the standard outcome conjunction $\bigoplus$.

$$\Gamma, \mathcal{P} \vDash \mathop{\&}_{X \in E} \varphi \qquad \text{iff} \qquad \Gamma, \mathcal{P} \vDash \bigoplus_{X \sim \mu} \varphi \quad \text{for some} \quad \mu \in \mathcal{D}\left(\llbracket E \rrbracket_{\mathsf{LExp}}(\Gamma)\right)$$

$$\lceil P \rceil * \varphi \dashv\vdash \lceil P \rceil *_{\mathsf{w}} \varphi \qquad \varphi *_{\mathsf{w}} \psi \vdash \varphi * \psi \qquad \frac{\varphi \vdash \varphi' \qquad \psi \vdash \psi'}{\varphi *_m \psi \vdash \varphi' *_m \psi'} \qquad \frac{\varphi \vdash \psi}{\&_{X \in E}\, \varphi \vdash \&_{X \in E}\, \psi}$$

$$\frac{Y \notin \mathsf{vars}(\varphi)}{\&_{X \in E}\, \varphi \vdash \&_{Y \in E}\, \varphi[Y/X]} \qquad \frac{\lceil \mathsf{supp}(d(E)) \subseteq E' \rceil}{\bigoplus_{X \sim d(E)} \varphi \vdash \&_{X \in E'}\, \varphi} \qquad \&_{X \in \{E\}}\, \varphi \dashv\vdash \varphi[E/X]$$

$$\frac{X \notin \mathsf{vars}(\psi)}{(\&_{X \in E}\, \varphi) * \psi \vdash \&_{X \in E}(\varphi * \psi)} \qquad \frac{X \notin \mathsf{vars}(\psi) \qquad \mathsf{precise}(\psi)}{\&_{X \sim d(E)}(\varphi * \psi) \vdash (\&_{X \sim d(E)}\, \varphi) * \psi}$$

$$\frac{X \notin \mathsf{vars}(\psi) \qquad \mathsf{convex}(\psi)}{\&_{X \in E}(\varphi *_{\mathsf{w}} \psi) \vdash (\&_{X \in E}\, \varphi) *_{\mathsf{w}} \psi} \qquad \frac{X \notin \mathsf{vars}(\varphi) \qquad \mathsf{convex}(\varphi)}{\&_{X \sim d(E)}\, \varphi \vdash \varphi}$$

Figure 5.4: Selected entailment laws with new assertions for nondeterministic reasoning.

5.3.3 Entailment Laws

Since the assertions that we have introduced are a conservative extension of the ones from Chapter 3, all of the entailment laws from Figure 3.3 carry over. We now present additional entailment laws relating to weak separation and nondeterministic outcome conjunctions.

First, we note that the new constructs are not precise (Definition 3.10), since they do not dictate the probabilities of all measurable events. However, they are both convex (Definition 3.11), similarly to how their probabilistic analogues are precise.

$$\frac{\mathsf{convex}(\varphi, \psi)}{\mathsf{convex}(\varphi *_{\mathsf{w}} \psi)} \qquad \frac{\mathsf{convex}(\varphi) \qquad \varphi \Rightarrow \lceil e \mapsto X \rceil}{\mathsf{convex}(\&_{X \in E}\, \varphi)}$$

New entailment laws are shown in Figure 5.4, which highlight the differences between the strong and weak separating conjunctions and the standard and nondeterministic outcome conjunctions.

First, the meaning of the strong and weak separating conjunctions are identical when one of the conjuncts is a pure assertion $\lceil P \rceil$, since pure assertions give vacuous independence guarantees (recall Section 3.1.3). In addition, strong separation can always be weakened to weak separation. Just as before, weakening can be applied under both weak and strong separating conjunctions, so the third rule is a generalized version of the one in Figure 3.3.

Nondeterministic separating conjunctions have similar properties to standard ones. Weakening applies under the modality, and the bound variable can be α-renamed. In addition, outcome conjunctions can be weakened to nondeterministic outcome conjunctions, that range over a superset of the support of the original outcome conjunction. We let supp be defined as follows:

$$\mathrm{supp}(\mathbf{Ber}\,(E)) \triangleq \begin{cases} \{0,1\} & \text{if } 0 < E < 1 \\ \{0\} & \text{if } E = 0 \\ \{1\} & \text{if } E = 1 \end{cases} \qquad \mathrm{supp}(\mathbf{geo}\,(E)) \triangleq \mathbb{N} \qquad \mathrm{supp}(\mathbf{unif}\,(E)) \triangleq E$$

A nondeterministic outcome conjunction over a singleton set $\{E\}$ is equivalent to the a single outcome $\varphi[E/X]$ with E substituted for X.

Terms can only distribute into outcome conjunctions if they are joined with a strong separating conjunction. Since weak separation does not preclude correlation, the implication below is invalid. On the left side, it is possible that $x = y$, since both are distributed in the same way and are not necessarily independent. However, on the right side, that distribution is precluded.

$$\left(\bigoplus_{X\sim\mathbf{Ber}\left(\frac{1}{2}\right)} \lceil x \mapsto X \rceil\right) *_{\mathrm{w}} \left(\bigoplus_{Y\sim\mathbf{Ber}\left(\frac{1}{2}\right)} \lceil y \mapsto Y \rceil\right) \quad \not\Rightarrow \quad \bigoplus_{X\sim\mathbf{Ber}\left(\frac{1}{2}\right)} \bigoplus_{Y\sim\mathbf{Ber}\left(\frac{1}{2}\right)} \lceil x \mapsto X \rceil *_{\mathrm{w}} \lceil y \mapsto Y \rceil$$

As before, a term ψ joined with a strong separating conjunction can be factored outside of a nondeterministic outcome conjunction as long as ψ is precise. If ψ is instead joined with a weak separating conjunction, then it only needs to be convex in order to be factored out. Finally, nondeterministic outcome conjunctions over a convex assertion φ can be folded into a single assertion φ as long as the bound variable X is not free in φ.

5.4 DEMONIC OUTCOME LOGIC

We now present Demonic Outcome Logic (dOL), an extension of pOL for reasoning about programs that are both probabilistic and nondeterministic. In dOL, nondeterminism is treated *demonically*, meaning that the postcondition holds regardless of how the adversarial nondeterminism is resolved in the program execution. There are two key differences in the definition of validity of dOL triples compared to pOL triples. First, the program semantics is interpreted in the convex

powerdomain rather than individual probability distributions. So, a universal quantifier is used to state that every distribution in the convex set of results satisfies the postcondition. The second difference is that the frame preservation property is now parameterized by a mode $m \in \{\mathrm{s}, \mathrm{w}\}$.

Definition 5.7 (Demonic Outcome Triples). $\vDash_m^{\text{DOL}} \langle \varphi \rangle\ C\ \langle \psi \rangle$ *iff for any* Γ, μ, *and probability spaces* $\mathcal{P}$, $\mathcal{P}_F$, *and* $\mathcal{P}' \in \mathcal{P} \diamond_m \mathcal{P}_F$ *such that* $\mathcal{P}' \preceq \mu$ *and* $\Gamma, \mathcal{P} \vDash \varphi$, *then:*

$$\forall \nu \in [\![C]\!]_C^\dagger(\mu). \qquad \exists \mathcal{Q}, \mathcal{Q}' \in \mathcal{Q} \diamond_m \mathcal{P}_F. \qquad \mathcal{Q}' \preceq \nu \quad \textit{and} \quad \Gamma, \mathcal{Q} \vDash \psi$$

Despite the differences between pOL and dOL, the meanings of two logics coincide when we restrict dOL to pCL programs (which do not have any nondeterministic constructs).

Theorem 5.8. *dOL is a conservative extension of pOL. That is, for any pCL program* $C \in \mathit{cmd}_P$:

$$\vDash_{\mathrm{s}}^{\text{DOL}} \langle \varphi \rangle\ C\ \langle \psi \rangle \qquad \textit{iff} \qquad \vDash^{\text{POL}} \langle \varphi \rangle\ C\ \langle \psi \rangle$$

Proof. With the mode specialized to s, $\vDash_{\mathrm{s}}^{\text{DOL}} \langle \varphi \rangle\ C\ \langle \psi \rangle$ is definitionally equivalent to:

$$\forall \Gamma, \mu, \mathcal{P}, \mathcal{P}_F.\ \mathcal{P} \otimes \mathcal{P}_F \preceq \mu \wedge \Gamma, \mathcal{P} \vDash \varphi \implies \forall \nu \in [\![C]\!]_C^\dagger(\mu).\ \exists \mathcal{Q}.\ \mathcal{Q} \otimes \mathcal{P}_F \preceq \nu \wedge \Gamma, \mathcal{Q} \vDash \psi$$

By Theorem 5.6, the above is equivalent to:

$$\forall \Gamma, \mu, \mathcal{P}, \mathcal{P}_F.\ \mathcal{P} \otimes \mathcal{P}_F \preceq \mu \wedge \Gamma, \mathcal{P} \vDash \varphi \implies \forall \nu \in \uparrow [\![C]\!]_{\mathcal{D}}^\dagger(\mu).\ \exists \mathcal{Q}.\ \mathcal{Q} \otimes \mathcal{P}_F \preceq \nu \wedge \Gamma, \mathcal{Q} \vDash \psi$$

Since $\mathcal{Q} \otimes \mathcal{P}_F \preceq \nu$, then $\nu(\bot) = 0$ for any $\nu \in \uparrow [\![C]\!]_{\mathcal{D}}^\dagger(\mu)$. Therefore, the upward closure must not add anything and therefore it is a singleton set: $\uparrow [\![C]\!]_{\mathcal{D}}^\dagger(\mu) = \left\{ [\![C]\!]_{\mathcal{D}}^\dagger(\mu) \right\}$. We can therefore rewrite the formula above to be the exact definition of a pOL triple $\vDash^{\text{POL}} \langle \varphi \rangle\ C\ \langle \psi \rangle$.

$$\forall \Gamma, \mu, \mathcal{P}, \mathcal{P}_F.\ \mathcal{P} \otimes \mathcal{P}_F \preceq \mu \wedge \Gamma, \mathcal{P} \vDash \varphi \implies \exists \mathcal{Q}.\ \mathcal{Q} \otimes \mathcal{P}_F \preceq [\![C]\!]_{\mathcal{D}}^\dagger(\mu) \wedge \Gamma, \mathcal{Q} \vDash \psi \qquad \square$$

Theorem 5.8 essentially implies that all of the inference rules from pOL carry over the dOL, albeit with the additional mode parameter. Of course, additional rules will be needed to reason about nondeterministic program constructs. We now present the proof rules of dOL and we will discuss soundness in Section 7.4. For the remainder of the chapter, we will omit the dOL

$$\frac{}{\vdash_m \langle \varphi \rangle\ \mathbf{skip}\ \langle \varphi \rangle}\textsc{Skip} \qquad \frac{\vdash_m \langle \varphi \rangle\ C_1\ \langle \vartheta \rangle \qquad \vdash_m \langle \vartheta \rangle\ C_2\ \langle \psi \rangle}{\vdash_m \langle \varphi \rangle\ C_1 \fatsemi C_2\ \langle \psi \rangle}\textsc{Seq}$$

$$\frac{\vdash_m \langle \varphi \rangle\ C_1\ \langle \psi_1 \rangle \quad \vdash_m \langle \varphi \rangle\ C_2\ \langle \psi_2 \rangle \quad \varphi \Rrightarrow \lceil e \mapsto E \rceil \quad \psi_1 \Rrightarrow \lceil e' \mapsto 1 \rceil \quad \psi_2 \Rrightarrow \lceil e' \mapsto 0 \rceil}{\vdash_m \langle \varphi \rangle\ C_1 \oplus_e C_2\ \langle \psi_1 \oplus_E \psi_2 \rangle}\textsc{Prob}$$

$$\frac{\varphi \Rrightarrow \lceil b \mapsto 1 \rceil \qquad \vdash_m \langle \varphi \rangle\ C_1\ \langle \psi \rangle}{\vdash_m \langle \varphi \rangle\ \mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \langle \psi \rangle}\textsc{IfT} \qquad \frac{\varphi \Rrightarrow \lceil b \mapsto 0 \rceil \qquad \vdash_m \langle \varphi \rangle\ C_2\ \langle \psi \rangle}{\vdash_m \langle \varphi \rangle\ \mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \langle \psi \rangle}\textsc{IfF}$$

$$\frac{\varphi \Rrightarrow \lceil e \mapsto E \rceil \wedge (\psi * \mathsf{own}(x))}{\vdash_m \langle \varphi \rangle\ x := e\ \langle \lceil x \mapsto E \rceil * \psi \rangle}\textsc{Assign} \qquad \frac{\varphi \Rrightarrow \lceil e \mapsto E \rceil \wedge (\psi * \mathsf{own}(x))}{\vdash_m \langle \varphi \rangle\ x :\approx d(e)\ \langle (x \sim d(E)) * \psi \rangle}\textsc{Samp}$$

$$\frac{\varphi' \Rrightarrow \varphi \qquad \vdash_m \langle \varphi \rangle\ C\ \langle \psi \rangle \qquad \psi \Rrightarrow \psi'}{\vdash_m \langle \varphi' \rangle\ C\ \langle \psi' \rangle}\textsc{Consequence} \qquad \frac{\vdash_m \langle \varphi \rangle\ C\ \langle \psi \rangle \quad \psi \Rrightarrow \lceil e \mapsto X \rceil}{\vdash_m \langle \bigoplus_{X \sim d(E)} \varphi \rangle\ C\ \langle \bigoplus_{X \sim d(E)} \psi \rangle}\textsc{Split1}$$

$$\frac{\vdash_m \langle \varphi \rangle\ C\ \langle \psi \rangle \quad \mathsf{convex}(\psi) \quad X \notin \mathsf{vars}(\psi)}{\vdash_m \langle \bigoplus_{X \sim d(E)} \varphi \rangle\ C\ \langle \psi \rangle}\textsc{Split2}$$

Figure 5.5: Inference rules that carry over to dOL from pOL.

superscript and simply write $\vDash_m \langle \varphi \rangle\ C\ \langle \psi \rangle$ to denote validity of a dOL triple. When $m = \mathrm{s}$, we will also omit the subscript and simply write $\vDash \langle \varphi \rangle\ C\ \langle \psi \rangle$.

The carryover rules from pOL are shown in Figure 5.5. The only difference between the new rules compared to the versions from Figure 4.1 is that the mode must match between the premises and the conclusion. Refer to Section 4.3 for an in depth description of the rules.

5.4.1 The Frame Rule

The Frame rule of dOL, shown in Figure 5.6, differs somewhat from that of pOL in that the type of separation between the local state and the frame is dictated by the triple's mode. The triple's mode can be altered with strengthening and weakening rules, also shown in Figure 5.6.

A strong triple can be weakened using Weaken, so that it only supports framing with sepa-

$$\frac{\vdash_m \langle \varphi \rangle\ C\ \langle \psi \rangle}{\vdash_m \langle \varphi *_m \vartheta \rangle\ C\ \langle \psi *_m \vartheta \rangle}\textsc{Frame} \qquad \frac{\vdash \langle \varphi \rangle\ C\ \langle \psi \rangle}{\vdash_w \langle \varphi \rangle\ C\ \langle \psi \rangle}\textsc{Weaken}$$

$$\frac{\vdash_w \langle \varphi \rangle\ C\ \langle \psi \rangle \qquad \mathsf{precise}(\psi)}{\vdash \langle \varphi \rangle\ C\ \langle \psi \rangle}\textsc{Strengthen}$$

Figure 5.6: Framing rules for dOL.

ration of state and does not guarantee that the program C preserves probabilistic independence between the local variables and the rest of the global state. Going in the opposite direction via the Strengthen rule additionally requires that the postcondition ψ is precise. If ψ is precise, then any nondeterminism in the command C is not measurable, and therefore the demonic adversary cannot induce any measurable correlation, allowing us to reenable the strong frame rule.

5.4.2 Nondeterminism Rules

The next set of new rules for dOL pertain to reasoning about nondeterminism, shown in Figure 5.7. As we already saw in Section 5.1.3, reasoning about nondeterminism is somewhat fraught, and care must be taken to design sound rules. For that reason, the rules for analyzing nondeterministic choice $C_1 \,\&\, C_2$ and nondeterministic assignment $x \leftarrow e$ differ from their probabilistic counterparts ($C_1 \oplus_e C_2$ and $x :\approx d(e)$) in two key ways. First, they require the resulting triples to only be weakly frame preserving, since the demonic adversary could induce a correlation with the rest of the global memory. Refer to Section 5.1.3 for a concrete example of induced correlation.

The second change is that the precondition must be a single pure assertion $\lceil P \rceil$. The restriction on the precondition ensures that the sequencing of probabilistic and nondeterministic choices are respected, so that if the precondition already represents a distribution of outcomes, then Split1 or Split2 must first be used to apply the inference about the nondeterministic choice to each

$$\frac{\vdash_w \langle \lceil P \rceil \rangle\ C_1\ \langle \psi \rangle \qquad \vdash_w \langle \lceil P \rceil \rangle\ C_2\ \langle \psi \rangle \qquad \mathsf{convex}(\psi)}{\vdash_w \langle \lceil P \rceil \rangle\ C_1 \,\&\, C_2\ \langle \psi \rangle}\textsc{Nondet}$$

$$\frac{\lceil P \rceil \Rightarrow \lceil e \mapsto E \rceil \wedge (Q * \lceil \mathsf{own}(x) \rceil) \qquad \mathsf{finite}(E)}{\vdash_w \langle \lceil P \rceil \rangle\ x \leftarrow e\ \langle \mathbin{\&}_{X \in E} \lceil Q * x \mapsto X \rceil \rangle}\textsc{NAssign}$$

$$\frac{\vdash_m \langle \varphi \rangle\ C\ \langle \psi \rangle \qquad \psi \Rrightarrow \lceil e \mapsto X \rceil}{\vdash_m \langle \mathop{\&}_{X \in E} \varphi \rangle\ C\ \langle \mathop{\&}_{X \in E} \psi \rangle}\textsc{NSplit1} \qquad \frac{\vdash_m \langle \varphi \rangle\ C\ \langle \psi \rangle \qquad \mathsf{convex}(\psi) \qquad X \notin \mathsf{vars}(\psi)}{\vdash_m \langle \mathop{\&}_{X \in E} \varphi \rangle\ C\ \langle \psi \rangle}\textsc{NSplit2}$$

$$\frac{\vdash_w \langle \mathbin{\&}_{X \in E} \lceil P \rceil \rangle\ C\ \langle \psi \rangle \qquad \lceil P \rceil \Rrightarrow e \mapsto X}{\vdash_w \langle \lceil \exists X \in E.\ P \rceil \rangle\ C\ \langle \psi \rangle}\textsc{Exists}$$

Figure 5.7: dOL rules for reasoning about nondeterminism.

outcome, such as in the example below:

$$\frac{\dfrac{}{\vdash_w \langle \lceil x \mapsto X \rceil \rangle\ y \leftarrow \{0,1\}\ \langle \mathbin{\&}_{Y \in \{0,1\}} \lceil x \mapsto X * y \mapsto Y \rceil \rangle}\textsc{NAssign}}{\vdash_w \langle \bigoplus_{X \sim \mathbf{Ber}(\frac{1}{2})} \lceil x \mapsto X \rceil \rangle\ y \leftarrow \{0,1\}\ \langle \bigoplus_{X \sim \mathbf{Ber}(\frac{1}{2})} \mathbin{\&}_{Y \in \{0,1\}} \lceil x \mapsto X * y \mapsto Y \rceil \rangle}\textsc{Split1}$$

Here, the $\&$ ends up nested inside of the $\bigoplus$, reflecting the fact that the probabilistic choice happened first, and that the nondeterministic selection occurs for each fixed probabilistic outcome. The Nondet rule additionally requires that the postcondition is convex, since in reality, it may not be a single state that satisfies $\lceil P \rceil$, but rather a distribution of states. In addition requires that the the expression E constitutes a finite set of values in line with the finite nondeterminism restriction of the convex powerdomain.

Recall from the previous section that weak triples can be converted to strong triples as long as the postcondition is precise. Since all pure assertions are precise, we can derive the following inference rule that produces a strong triple for nondeterministic assignment.

$$\frac{\mathsf{finite}(E)}{\vdash \langle \lceil x \mapsto - \rceil \rangle\ x \leftarrow E\ \langle \lceil x \in E \rceil \rangle}$$

In Section 5.5, we will see more involved ways in which Strengthen is applied after a longer sequence of steps.

Next, the NSplit1 and NSplit2 enable case analysis on nonderministic outcomes. These two rules mirror their probabilistic counterparts (Split1 and Split2), which were explained in Section 4.3.2. The splitting rules do not require triples to be weakly frame preserving.

The final rule, Exists, pertains to moving between representations of nondeterminism. Indeed, at first glance the assertions $\&_{X \in E} \lceil P \rceil$ and $\lceil \exists X \in E.\ P \rceil$ may appear to be expressing the same thing, but in the former assertion the different outcomes are measurable (even if the probabilities are not known), whereas in the latter the outcomes are not necessarily measurable. As a result, the assertions below have very different meaning:

$$\left(\&_{X \in E} \lceil x \mapsto X \rceil \right) * \left(y \sim \mathbf{Ber}\left(\tfrac{1}{2}\right) \right) \qquad\qquad \lceil \exists X \in E.\ x \mapsto X \rceil * \left(y \sim \mathbf{Ber}\left(\tfrac{1}{2}\right) \right)$$

Although neither assertions dictates the probability of x taking on various values, the assertion on the left stipulates that x and y are probabilistically independent, whereas x and y may be correlated according to the assertion on the right. The problem is that, even with a precondition of $\lceil \exists X \in E.\ P \rceil$, we may need to do case analysis on the values of the bound variable X, for example, when the control flow of the program depends on X.

The Exists rule allows us to move between the $\&_{X \in E} \lceil P \rceil$ and $\lceil \exists X \in E.\ P \rceil$ representations in the precondition of a triple, so that we can then use NSplit1 or NSplit2 to do case analysis on the variable X. The conversion is only valid under weak frame preservation, since the premise of the rule does not require X to be independent of external global state when joined with a strong separating conjunction, whereas the premise does require independence.

5.4.3 Loops and Almost Sure Termination

Recall that in pOL, the BoundedRank rule allowed us to prove almost sure termination for a loop given a loop invariant φ and a bounded integer rank $\ell \le R \le h$ such that the rank strictly decreases on each iteration with probability p. Compared to pOL, we generalize the dOL loop

$$\frac{\vdash_m \langle \varphi * \lceil R = N > \ell \rceil \rangle\ C\ \langle \left(\&_{R=\ell}^{N-1}\ \varphi \right) \oplus_{\geq p} \left(\&_{R=N}^{h}\ \varphi \right) \rangle \quad \begin{array}{c} \varphi \Rightarrow \lceil \ell \leq R \leq h \rceil \\ \varphi * \lceil R = \ell \rceil \Rightarrow \lceil b \mapsto 0 \rceil \\ \varphi * \lceil R > \ell \rceil \Rightarrow \lceil b \mapsto 1 \rceil \end{array} \quad \begin{array}{c} N \notin \mathrm{vars}(\varphi) \\ 0 < p \leq 1 \\ \mathrm{precise}(\varphi[\ell/R]) \end{array}}{\vdash_m \langle\ \&_{R=\ell}^{h}\ \varphi \rangle\ \textbf{while}\ b\ \textbf{do}\ C\ \langle \varphi[\ell/R] \rangle}\ \textsc{BoundedRank}$$

Figure 5.8: The loop analysis rule of dOL.

rule to account for the fact that the rank may be affected by nondeterminism too. The new rule is shown in Figure 5.8

The introduction of $\&$ into the assertion language means that we can remove the explicit distributions d, d_1, and d_2, and instead use $\&$ to state that the rank is distributed in some unspecified way. As such, the rule states that as long as the rank is initially distributed between ℓ and h, and that the rank strictly decreases with probability at least $p > 0$ on each iteration, then the loop will almost surely terminate.

5.5 EXAMPLES

To demonstrate the nondeterministic reasoning principles of dOL, we turn to the running examples first introduced in Section 4.4. The examples are now modified to include demonic choice in addition to the prior random behavior.

5.5.1 Adversarial Random Walk

The first example is an adversarial variant of the random walk from Section 4.4.1. The goal of the random walk is still to reach the origin, and the agent still moves toward the origin with probability $\frac{1}{2}$. However, now, in the case where the agent does not move toward the origin, the agent's position gets reset to a demonically chosen location between x and z.

$$\begin{array}{l} \textbf{while}\ x > 0\ \textbf{do} \\ \quad x := x - 1 \oplus_{\frac{1}{2}} x \leftarrow \{x, .., \max(x, z)\} \end{array}$$

The demonic variant of the random walk still almost surely terminates. Since the adversary's reset is bounded by z, the adversary will reach the origin with probability at least $\frac{1}{2^z}$ at any point

in time, which is nonzero, and therefore the program almost surely terminates. We will now prove that fact rigorously using dOL.

We start with the precondition $\lceil x \mapsto X * z \mapsto Z \rceil$, *i.e.,* that x and z have some universally quantified values, and we will separately assume that X and Z are natural numbers. Next, we need a loop invariant φ. Similar to the prior version, the invariant states that the rank R is simply the value of x, but now R is bounded between 0 and the max of X and Z. In addition, the rank will decrease on each iteration with probability at least $\frac{1}{2}$.

$$\varphi \triangleq \lceil x \mapsto R * z \mapsto Z * 0 \leq R \leq \max(X, Z) \rceil \qquad\qquad p \triangleq \frac{1}{2}$$

The derivation is shown in Figure 5.9. The first aspect to note is that the nondeterministic assignment inside of the loop body forces us to work with a weakly frame preserving triple. While the loop is executing, the distribution over values of x depends on the adversary's behavior, and therefore it is not necessarily probabilistically independent from the distribution over other variables in the global memory. However, the adversary's behavior cannot affect the probability of eventual termination. As such, at the end of the derivation we get the precise postcondition $\lceil x \mapsto 0 \rceil$, *i.e.,* the program almost surely terminates in a state where x has value 0. Therefore, we can use Strengthen to reenable strong frame preservation.

The remainder of the proof is quite mechanical. After applying BoundedRank, we apply Prob to split up the two random outcomes. The first case is dispatched with a simple application of the Assign rule and the second case is dispatched with NAssign.

5.5.2 The Adversarial von Neumann Trick

The second of the two running examples is the von Neumann [1951] trick, which is a randomness preserving protocol for simulating a fair coin given a biased coin. In Section 4.4.2, we saw a correctness proof for the classical version of the protocol, where the coin's bias is fixed. We now present an adversarial version where the bias is nondeterministically chosen on each round.

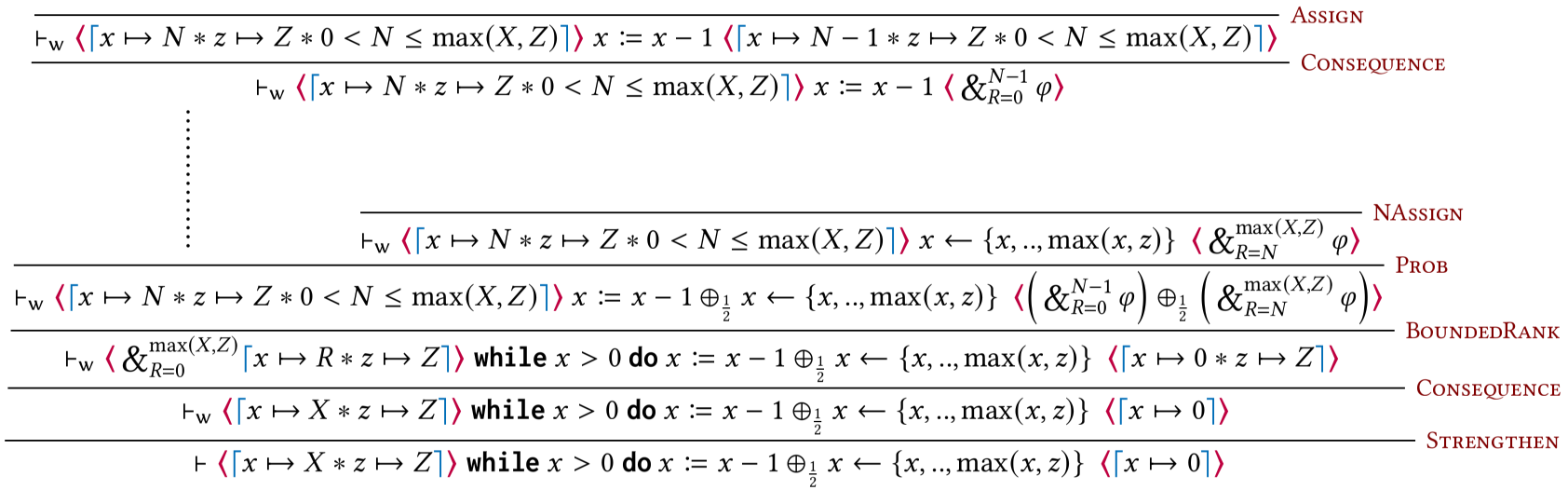


Figure 5.9: Almost sure termination proof for the adversarial random walk

The new variant of the program is shown below. The only change is that p is no longer a constant, but rather it is chosen nondeterministically from the interval $[\varepsilon, 1-\varepsilon]_\Delta$, where Δ is a finite step size, making the set finite and $0 < \varepsilon \leq \frac{1}{2}$ is an arbitrarily small probability.

$$
\begin{array}{l}
x \coloneqq 0 \,⨟ \\
y \coloneqq 0 \,⨟ \\
\textbf{while } x = y \textbf{ do} \\
\quad p \leftarrow [\varepsilon, 1-\varepsilon]_\Delta \,⨟ \\
\quad x :\approx \mathbf{Ber}\,(p) \,⨟ \\
\quad y :\approx \mathbf{Ber}\,(p)
\end{array}
$$

The derivation for the adversarial von Neumann program is shown in Figure 5.10. Since the first two lines obviously leave us in a state where $\lceil x \mapsto 0 * y \mapsto 0 \rceil$, we omit those commands and complete the derivation only for the while loop. Just like the adversarial random walk, the nondeterminism inside the loop body forces us to use a weakly frame preserving triple. The adversary can influence the probability of terminating on each round, therefore the local state at any intermediary stage is not necessarily probabilistically independent from any other global state. However, we will prove that $x \sim \mathbf{Ber}\left(\frac{1}{2}\right)$ upon termination, which is precise, therefore we can use Strengthen at the end to regain strong frame preservation.

Before we can analyze the while loop, we first need a loop invariant and ranking function. As before, the rank is bounded between 0 and 1. When the rank is 0, $x \neq y$ and the loop terminates. In that case, x is also uniformly distributed. When the rank is 1, then $x = y$ and the loop keeps going. The only difference from Section 4.4.2 is that now we add $\mathsf{own}(p)$ to φ_1.

$$\varphi \triangleq (\varphi_0 * \lceil R = 0 \rceil) \vee (\varphi_1 * \lceil R = 1 \rceil)$$

$$\varphi_0 \triangleq \bigoplus_{X \sim \mathbf{Ber}\left(\frac{1}{2}\right)} \lceil x \mapsto X * y \mapsto \neg X \rceil \qquad\qquad \varphi_1 \triangleq \lceil (x = y) \mapsto 1 * \mathsf{own}(p) \rceil$$

Clearly, $\lceil x \mapsto - * y \mapsto 0 * \mathsf{own}(p) \rceil \Rightarrow \varphi_1 \Rightarrow \&^1_{R=0}\, \varphi$, therefore we can use BoundedRank to handle the while loop. In the premise, we start by using Seq to break up the sequential composition. An application of NAssign gives us a $\&$ ranging over the possible values for p.

Next, before analyzing the remaining sequence of two sampling operations, we use NSplit2 to do case analysis on the exact value of p. That allows us to mechanically derive the postcondition

$x \sim \mathbf{Ber}(Z) * y \sim \mathbf{Ber}(Z) * p \mapsto Z$ after the sampling operations execute (note that $\varepsilon \leq Z \leq 1-\varepsilon$, although we omit that detail from Figure 5.10 to conserve space).

The final step is to use a consequence to derive the desired postcondition for the premise of BoundedRank. We start by expanding the outcome conjunctions and distributing the all of the pure assertions to the inside.

$$x \sim \mathbf{Ber}(Z) * y \sim \mathbf{Ber}(Z) * p \mapsto Z$$
$$\Longrightarrow \bigoplus_{X \sim \mathbf{Ber}(Z)} \bigoplus_{Y \sim \mathbf{Ber}(Z)} \lceil x \mapsto X * y \mapsto Y * \mathsf{own}(p) \rceil$$

Next, we note that there are two outcomes where $x \neq y$: either $X = 1$ and $Y = 0$ (with probability $Z \cdot (1 - Z)$) or $X = 0$ and $Y = 1$ (with probability $(1 - Z) \cdot Z$). Therefore, the total probability that $x \neq y$ is $2 \cdot Z \cdot (1 - Z)$, and within that outcome, it is equally likely that $x \mapsto 1$ or $x \mapsto 0$.

$$\Longrightarrow \left(\bigoplus_{X \sim \mathbf{Ber}\left(\frac{1}{2}\right)} \lceil x \mapsto X * y \mapsto \neg X \rceil \right) \oplus_{2Z(1-Z)} \lceil (x = y) \mapsto 1 * \mathsf{own}(p) \rceil$$

The two operands of the $\oplus$ are exactly equal to φ_0 and φ_1, so we can fold them according to their definitions. Finally, since we know that $\varepsilon \leq Z \leq 1-\varepsilon$, then it must be that $2 \cdot Z \cdot (1-Z) \geq 2 \cdot \varepsilon \cdot (1-\varepsilon)$, and therefore we can replace the exact outcome conjunction with an approximate one in order.

$$\Longrightarrow \varphi_0 \oplus_{\geq 2\varepsilon(1-\varepsilon)} \varphi_1$$

Since the assertion above no longer depends on Z, we apply NSplit2 to complete the proof.

5.6 RELATED WORK

Semantic Domains for Probabilistic Concurrency. Powerdomains are a domain theoretic tool for reasoning about looping nondeterministic programs, which provide a domain in which loops can be interpreted as fixed points. Powerdomain revolve around defining orders over sets of states to show that iterated actions eventually converge. Given a partially ordered domain of

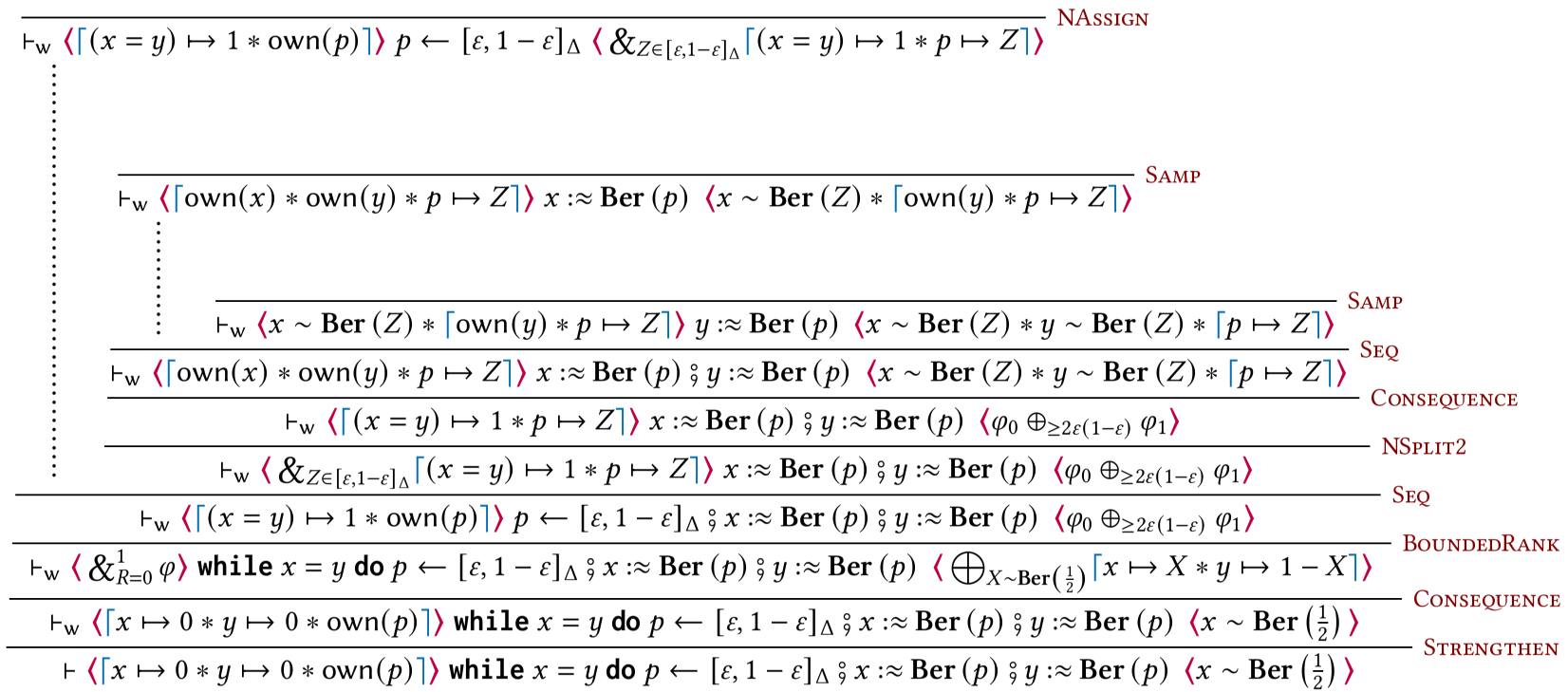


Figure 5.10: Correctness proof for the adversarial von Neumann trick.

program states $\langle \Sigma, \leq \rangle$, there are three typical choices for orders over sets of states, known as the Hoare, Smyth [1978], and Egli-Milner orders, defined below:

$$
\begin{array}{lll}
S \sqsubseteq_{\mathsf{H}} T & \text{iff} & \forall \sigma \in S.\ \exists \tau \in T.\ \sigma \leq \tau \\
S \sqsubseteq_{\mathsf{S}} T & \text{iff} & \forall \tau \in T.\ \exists \sigma \in S.\ \sigma \leq \tau \\
S \sqsubseteq_{\mathsf{EM}} T & \text{iff} & S \sqsubseteq_{\mathsf{H}} T \quad \text{and} \quad S \sqsubseteq_{\mathsf{S}} T
\end{array}
$$

In general, none of these relations are antisymmetric, making them *preorders*, whereas domain theoretic tools for finding fixed points operate on *partial orders*. So the sets representing the program semantics must be closed in order to obtain a proper domain. This closure operation loses precision of the semantics, incorporating additional possibilities which are not always intuitive.

The Hoare order requires a down-closure, essentially meaning that nontermination may always be an option. The Hoare order is therefore a good choice for partial correctness, where we only wish to determine what happens *if* the program terminates, as in Hoare Logic. The Smyth [1978] order, which we use in our definition of the convex powerdomain, requires an upwards closure, so that nontermination becomes erratic behavior. The upward closure makes the Smyth order appropriate for total correctness [Manna and Pnueli 1974], which is concerned only with terminating programs where erratic behavior does not arise.

In the Egli-Milner case (and the associated Plotkin Powerdomain [1976]) the more precise, but also less intuitive Egli-Milner closure is used. McIver and Morgan [2001] created a denotational model where fixed points are taken with respect to the Egli-Milner order rather than the Smyth [1978] one. As such, they require the domain of computation to be Egli-Milner closed, which means that $S = \uparrow S \cap \downarrow S$. Unlike up-closedness (required for the Smyth approach), which is preserved by all the operations in Figure 5.2, the semantics of McIver and Morgan [2001] must take the Egli-Milner closure after nondeterministic and probabilistic choice and after sequential composition, making the model more complex and adding outcomes that do not have an obvious operational meaning. Refer to Tix et al. [2009]; Keimel and Plotkin [2017] for a more complete exploration of that approach.

Let us examine the semantics of a coin flip in order to demonstrate why the Smyth order is

preferable to Hoare. The variable x is assigned the values 1 or 0 each with probability $\frac{1}{2}$. So, the result of running the program is a singleton set containing the aforementioned distribution.

$$[\![x :\approx \mathbf{Ber}\left(\tfrac{1}{2}\right)]\!](\sigma) = \left\{ \begin{bmatrix} \sigma[x \coloneqq 1] & \mapsto & \frac{1}{2} \\ \sigma[x \coloneqq 0] & \mapsto & \frac{1}{2} \end{bmatrix} \right\}$$

If we were to use the Hoare powerdomain, then we would need to down-close the set, adding all smaller distributions too. That down closure not only includes nontermination as a possible outcome, but also prevents us from even determining that x is 1 or 0 with equal probability.

$$[\![x :\approx \mathbf{Ber}\left(\tfrac{1}{2}\right)]\!](\sigma) = \left\{ \begin{bmatrix} \sigma[x \coloneqq 1] & \mapsto & p \\ \sigma[x \coloneqq 0] & \mapsto & q \\ \bot & \mapsto & 1-p-q \end{bmatrix} \middle| \quad p \leq \frac{1}{2},\ q \leq \frac{1}{2} \right\}$$

Varacca's [2002] indexed valuation model was formulated in the Hoare powerdomain, and the loss of precision is reflected in the associated adequacy theorems. In particular, Varacca's Proposition 6.10 shows that the denotational model includes outcomes that may not be possible according to the associated operational model. By contrast, the up-closure that is required by the Smyth order adds nothing for the simple coin flip; the semantics is already a full distribution and therefore there are no distributions larger than it. We can therefore conclude that the two outcomes occur with probability exactly $\frac{1}{2}$ and that nontermination is impossible, as desired.

Powerdomains demonstrate that the notion of partial correctness (as embodied by the Hoare order) does not make much sense in probabilistic settings, since it translates to uncertainty about the minimum probability of an event. Total correctness, on the other hand, does make sense, and corresponds to the notion of *almost sure termination*, which is a property of great interest in probabilistic program analysis [McIver et al. 2018; Chakarov and Sankaranarayanan 2013].

The problem with the Smyth powerdomain is that it is not a DCPO in the presence of unbounded nondeterminism [Apt and Plotkin 1986; Søndergaard and Sestoft 1992], which is why He et al. [1997] instead use the Knaster-Tarski theorem to guarantee the fixed point existence via transfinite iteration, which only requires monotonicity and not Scott continuity. The main shortcoming of He et al.'s approach is that it did not guarantee non-emptiness of the set of result distributions, meaning that some programs may have vacuous semantics.

In response, Morgan et al. [1996a] added the additional requirement that the domain only include topologically closed sets (Morgan et al. called this property *Cauchy closure*). Topological closure ensures that no programs are modeled as empty sets, but it also prevents commands from exhibiting unbounded nondeterminism. For example, it is not possible to represent a program $x \coloneqq \bigstar$, which nondeterministically selects a value for x from the natural numbers.

Further work by Tix [1999, 2000], Tix et al. [2009], and Keimel and Plotkin [2017] has extended convex powerdomain semantics using powerdomains other than the Smyth one, and allow for continuous distributions rather than only discrete ones.

Monads for Probabilistic Nondeterminism. Varacca [2002, 2003] introduced powersets of *indexed valuations*. An indexed valuation behaves similarly to a distribution, but the idempotence property is removed, *i.e.*, $X \oplus_p X \neq X$. As shown by Varacca and Winskel [2006], a powerset of indexed valuations has a Beck [1969] distributive law, and is therefore a monad. However, indexed valuations are difficult to work with since equivalence is taken modulo renaming of the indices.

Varacca [2002, Theorem 6.5] proved that denotational models based on indexed valuations are equivalent to operational models in which a *deterministic* scheduler resolves the nondeterminism. With the goal in mind of modeling *adversarial* nondeterminism, convex sets provide correctness guarantees against a stronger probabilistic scheduler, giving us more robust guarantees.

An alternative approach is to flip the order of composition and work instead with distributions of nondeterministic outcomes. While the distribution monad does not compose with powerset, it does compose with multiset, as shown by Jacobs [2021] and further explored by Kozen and Silva [2024]. The downside is that the multisets must be finite, but it is easy to construct programs that reach infinitely many nondeterministic outcomes via while loops. So the distribution of mutiset model cannot be used to represent arbitrary programs from the language in Section 6.2. The use of multiset instead of powerset is again an instance of removing an idempotence law, this time for nondeterminism: $X \,\&\, X \neq X$. Indeed, idempotence is the key reason why no distributive law exists in both cases [Parlant 2020; Zwart and Marsden 2019; Zwart 2020].

Weakest Pre-expectations. From the start, weakest pre-expectations (wp) supported nondeterminism; in fact, wp emerged from a line of work on semantics for randomized nondeterministic programs [He et al. 1997; McIver and Morgan 2001; Morgan et al. 1996b]. Nondeterminism is handled by lower-bounding expectations, corresponding to larger expected values being *better*. An *angelic* variant can alternatively be used for upper bounds.

As noted by Kaminski [2019, §2.3.3], the choice of either upper or lower bounding the expected values is *"extremal"*—it forces a view where expectations must be either maximized or minimized, as opposed to our approach where multiple outcomes can be represented in one specification. However, reasoning about outcomes and expectations are not mutually exclusive; Barthe et al. [2018, Theorem 1] showed how to embed a wp calculus in a probabilistic program logic. A similar construction could be explored in Demonic Outcome Logic.

CHAPTER 6

DENOTATIONAL SEMANTICS FOR PROBABILISTIC CONCURRENT PROGRAMS

Before extending dOL with concurrency reasoning principles, we must take a detour to discuss the semantics of programs that are both probabilistic and concurrent. Although the approach laid out in Section 2.1, where the semantics follows from a monad with a DCPO structure, captures a wide variety of effects, concurrency is fundamentally different because running two programs in parallel can completely change their respective behaviors.

As a result, this chapter develops an underlying semantic structure called pomsets with formulae, which capture the meaning of concurrent randomized programs in a compositional way. We then provide a linearization procedure, which returns the set of all distributions that can be obtained by the different interleavings of the parallel threads. Finally, we show that the linearization semantics for pdCL programs (without parallel composition) yields the exact same convex powerdomain semantics $[\![-]\!]_C$ from the previous chapter, allowing us to extend dOL to handle concurrency in the next chapter.

6.1 OVERVIEW

When two programs are composed in parallel, the resulting behavior can be completely different from the behavior of the programs when run in isolation. As an example of behavioral changes, consider the following concurrent program.

$$\begin{array}{l} x \coloneqq 0 \fatsemi \\ \textbf{if } x = 0 \textbf{ then} \\ \quad y \coloneqq 0 \\ \textbf{else} \\ \quad y \coloneqq 1 \end{array} \;\Big\|\; x \coloneqq 1$$

It is relatively easy to see that running the left thread in isolation will always yield a state where x and y are both 0. In addition, the right thread will set x to 1 and leave y unchanged. However, the composite program has an interleaving where $x \coloneqq 1$ is executed between $x \coloneqq 0$ and the guard

$$
\begin{array}{lll}
cmd_{\text{PC}} \ni C ::= & \textbf{skip} & \textit{(No-op)} \\
& \mid C_1 \fatsemi C_2 & \textit{(Sequential Composition)} \\
& \mid C_1 \oplus_p C_2 & \textit{(Probabilistic Choice)} \\
& \mid C_1 \,\&\, C_2 & \textit{(Demonic Choice)} \\
& \mid C_1 \parallel C_2 & \textit{(Parallel Composition)} \\
& \mid \textbf{if } b \textbf{ then } C_1 \textbf{ else } C_2 & \textit{(Conditional Branching)} \\
& \mid \textbf{while } b \textbf{ do } C & \textit{(Loops)} \\
& \mid a & \textit{(Basic Actions } a \in act\textit{)}
\end{array}
$$

Figure 6.1: Syntax of the Probabilistic Concurrent Command Language (PCCL). The new addition over PDCL is highlighted in yellow. As before, tests $b \in test$ and expressions $e \in exp$ are discussed in Section 2.1.1, distributions d are from Figure 2.3, and actions are the same from Figure 5.1.

of the if statement, which will cause y to be set to 1, even though that was not an observable behavior in either thread on its own. Therefore, we cannot capture the semantics as a function $[\![C]\!] : \mathsf{Mem}[V] \to \mathcal{C}(\mathsf{Mem}[V])$, but rather, we need a more sophisticated semantic structure, which tracks the dependencies between all of the atomic actions in the program.

6.1.1 Pomsets with Formulae

Our goal is to develop a denotational model to capture concurrency in a manner that is compatible with the convex powerdomain. More concretely, we need a compositional way to model the semantics the Probabilistic Concurrent Command Language (PCCL) shown in Figure 6.1.

Partially ordered multisets (pomsets) are the predominant denotational technique for modeling concurrent programs [Gischer 1988; Pratt 1976; Brookes 2004; Grabowski 1981]. Pomsets use a partial order to denote *causality* between the actions in a concurrent program, allowing them to encode the two types of composition shown below.

$$
[\![a_1 \fatsemi a_2]\!]_{pom} = \begin{array}{c} a_2 \\ \uparrow \\ a_1 \end{array} \qquad\qquad [\![a_1 \parallel a_2]\!]_{pom} = \begin{array}{ccc} a_1 & & a_2 \\ & \nwarrow \; \nearrow & \\ & \bullet & \end{array}
$$

Sequential composition $a_1 \fatsemi a_2$ results in a causality $a_1 \to a_2$, meaning that a_1 must execute before

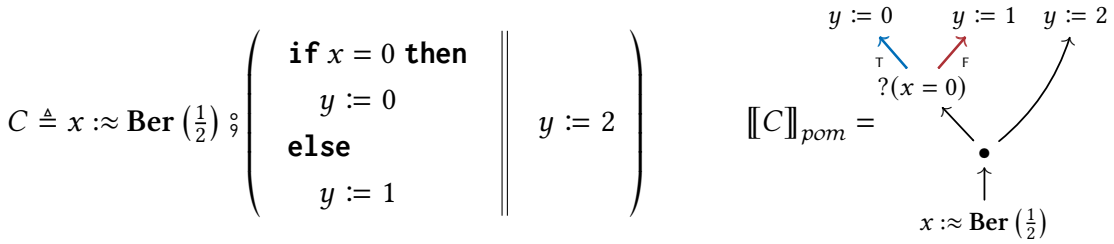


Figure 6.2: A randomized concurrent program (left) and its associated semantic structure (right).

a_2. By contrast, the parallel composition $a_1 \parallel a_2$ results in both actions stemming from a single root with no causality between them, meaning that a_1 and a_2 can execute in any order.

The problem with standard pomsets is that they cannot properly model control flow that depends on probabilistic computation, which is present in the program on the left of Figure 6.2. In that program, the result of a fair coin flip is used to control the flow of execution in one of two parallel branches. The key issue is that standard pomsets do not track control flow, and instead separate each computation branch into an individual trace. That approach is problematic in cases like Figure 6.2, since no trace has enough information to interpret all of the random outcomes. For a more complete counterexample, refer to Zilberstein et al. [2025a, §2].

We introduce a new semantic structure called *pomsets with formulae* to address the issue of randomized control flow. The key innovation is the use of Boolean formulae to capture guarded branching. As shown on the right of Figure 6.2, one single structure describes all the possible program traces, avoiding the need to separate traces in a way that is incompatible with probabilistic computation. Formulae show up in the new arrow types $b \xrightarrow{\mathsf{T}} a_1$ and $b \xrightarrow{\mathsf{F}} a_2$, which state that a_1 (resp. a_2) can only be scheduled if the test b evaluates to true (resp. false).

6.1.2 Unbounded Looping and Domain Theory

Combining all program traces into a single structure introduces challenges when modeling unbounded loops, such as the random walk and von Neumann trick programs from Chapters 4 and 5. Instead of recording an infinite set of *finite* traces, we now need a single infinite structure.

Although standard domain theoretic techniques allow us to construct infinite structures via fixed points, more work is needed to ensure that pomsets with formula form a DCPO.

Following from the Kleene Fixed Point Theorem, infinite pomsets will be constructed as the supremum of their finite unrollings, shown below.

$$[\![\textbf{while } b \textbf{ do } a]\!]_{pom} \quad = \quad \sup\left\{ \bot, \; \begin{array}{c} \bot \\ \uparrow \\ a \\ {\scriptstyle \mathsf{T}}\nwarrow \quad \nearrow{\scriptstyle \mathsf{F}} \\ b \quad \bullet \end{array}, \; \begin{array}{c} \bot \\ \uparrow \\ a \\ {\scriptstyle \mathsf{T}}\nwarrow \quad \nearrow{\scriptstyle \mathsf{F}} \\ b \quad \bullet \\ \uparrow \\ a \\ {\scriptstyle \mathsf{T}}\nwarrow \quad \nearrow{\scriptstyle \mathsf{F}} \\ b \quad \bullet \end{array}, \cdots \right\} \quad = \quad \begin{array}{c} \vdots \\ \uparrow \\ a \\ {\scriptstyle \mathsf{T}}\nwarrow \quad \nearrow{\scriptstyle \mathsf{F}} \\ b \quad \bullet \\ \uparrow \\ a \\ {\scriptstyle \mathsf{T}}\nwarrow \quad \nearrow{\scriptstyle \mathsf{F}} \\ b \quad \bullet \end{array}$$

Here, a new node label $\bot$ indicates that a computation is unfinished, and each pomset in the sequence expands a $\bot$ node to include an additional execution of the loop body. Although Meyer and de Vink [1989] had previously defined a DCPO structure for standard pomsets, the inclusion of formulae makes the construction more complex.

6.1.3 Linearization and Adequacy

Pomsets with formulae provide a compositional model for the semantics of randomized concurrent programs, but they cannot answer questions about the probabilities of outcomes that result from running the program. For example, in the program from Figure 6.2, we may wish to know the probability that y takes on the value 0, 1, or 2.

We determine those probabilities via *linearization* $\mathcal{L}\colon pom \to \mathsf{Mem}[V] \to C(\mathsf{Mem}[V])$, a function that interprets a pomset as a state transformer, mapping a start state to the convex set of end states obtained by different interleavings of the concurrent actions. Linearizing $[\![C]\!]_{pom}$ from Figure 6.2, we get the following set of distributions:

$$\mathcal{L}([\![C]\!]_{pom})(\sigma) = \left\{ \left[\begin{array}{lcl} s[x \coloneqq 0, y \coloneqq 0] & \mapsto & \frac{1}{2}p \\ s[x \coloneqq 0, y \coloneqq 2] & \mapsto & \frac{1}{2}(1-p) \\ s[x \coloneqq 1, y \coloneqq 1] & \mapsto & \frac{1}{2}q \\ s[x \coloneqq 1, y \coloneqq 2] & \mapsto & \frac{1}{2}(1-q) \end{array} \right] \;\middle|\; 0 \le p, q \le 1 \right\}$$

Linearization provides new insights about the program, which were not obvious in the pomset structure. After executing the coin flip, the scheduler can pick the order of updates to y in the two threads. Since we are interpreting the program in the convex powerdomain, the scheduler is allowed to make its choice probabilistically, hence we have two probabilities, p and q, which correspond to the different traces that result from the initial coin flip. The scheduler cannot make y equal to 0 or 1 with probability greater than $\frac{1}{2}$, but it can make $y = 2$ with any probability.

Linearization also makes it possible to compare pomset with formulae semantics with the standard powerdomain semantics from Section 5.2. In fact, for any PDCL program $C \in cmd_{\text{PD}}$, we prove that $\mathcal{L}(\llbracket C \rrbracket_{pom}) = \llbracket C \rrbracket_C$. As a consequence, the DOL inference rules for non-parallel commands will directly carry over to the concurrent setting.

6.2 SEMANTIC STRUCTURES

We will now define the mathematical structures that will form the basis of our semantic model. As discussed in Section 6.1, the semantic structures must account for both the deterministic and the parallel branching behavior of the PCCL language from Figure 6.1.

6.2.1 Labelled Partial Orders with Formulae

In order to formally represent mathematical objects like the ones in Figure 6.2, each node must have a unique identifier, so that we can describe its relationship to the other nodes. Using the node's label (*e.g.*, the actions and tests in Figure 6.2) is not sufficient, since each label may appear multiple times. We call the new structure a *labelled partial order with formulae* (LPOF).

As we already saw in Figure 6.2, the labels in an LPOF consist of actions, tests, and no-ops ($\bullet$). In order to account for nondeterministic and probabilistic branching, we extend the set of test labels to include symbols & and $\oplus_e$. As we saw in Section 6.1.2, $\bot$ is also label that represents an incomplete computation, which is used to express finite approximations of loops.

$$\mathit{label} \ni \ell ::= a \mid t \mid \bullet \mid \bot \qquad \mathit{etest} \ni t ::= b \mid \& \mid \oplus_e \qquad (a \in \mathit{act}, b \in \mathit{test}, e \in \mathit{exp})$$

Labels form a pointed DCPO, with the flat order $\leq \subseteq \mathit{label} \times \mathit{label}$ shown below.

$$\ell \leq \ell' \qquad \text{iff} \qquad \ell = \bot \quad \text{or} \quad \ell = \ell'$$

Let *nodes* be a countable universe of identifiers. Control flow restrictions are expressed using Boolean formulae over nodes, with the following grammar:

$$\mathit{form} \ni \psi ::= \mathbf{true} \mid \mathbf{false} \mid \psi_1 \wedge \psi_2 \mid \psi_1 \vee \psi_2 \mid \neg\psi \mid x \qquad (x \in \mathit{nodes})$$

Formulae are interpreted by valuations $v : \mathit{nodes} \to \mathbb{B}$. The satisfaction relation, written $v \vDash \psi$, is defined in the standard way. The variables of a formula ψ, written $\mathsf{vars}(\psi)$, are those nodes that appear in ψ. We write $\mathsf{sat}(\psi)$ iff there exists a valuation v such that $v \vDash \psi$; $\psi \Rightarrow \psi'$ iff $v \vDash \psi'$, for every $v \vDash \psi$; and $\psi \Leftrightarrow \psi'$ iff $\psi \Rightarrow \psi'$ and $\psi' \Rightarrow \psi$. Finally, we define the level of a poset element x, $\mathsf{lev}(x)$, which denotes the maximum length of a chain ending at x [Meyer and de Vink 1989].

Definition 6.1 (Level)**.** *Given a strict poset $\langle X, < \rangle$, define* $\mathsf{lev} : X \to \mathbb{N}^\infty$*:*

$$\mathsf{lev}(x) \triangleq \sup \{n \mid \exists x_0, \ldots, x_n \in X.\ x_n = x \ \wedge\ \forall i < n.\ x_i < x_{i+1}\}$$

The inverse image $\mathsf{lev}^{-1} : \mathbb{N} \to 2^X$ *is defined* $\mathsf{lev}^{-1}(n) \triangleq \{x \in X \mid \mathsf{lev}(x) = n\}$.

With those preliminary definitions, we can now define the LPOF structure.

Definition 6.2 (LPOF)**.** *A* labelled partial order with formulae *(LPOF) $\alpha = \langle N, <, \lambda, \varphi \rangle \in \mathit{lpo}$ is a 4-tuple where:*

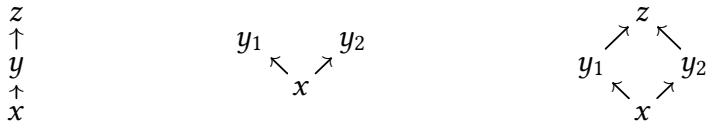
1. *$N \subseteq \mathit{nodes}$ is a countable set of nodes;*
2. *$\langle N, < \rangle$ is a (strict) poset such that:*
   (a) *it is finitely preceded, that is $|{\downarrow}x| < \infty$, for every $x \in N$;*
   (b) *every level has a finite number of nodes,* i.e., *$|\mathsf{lev}^{-1}(n)| < \infty$ for all $n \in \mathbb{N}$; and*
   (c) *it is single-rooted, that is: there exists an $x \in N$ such that $x < y$ for all $y \in N \setminus \{x\}$.*
3. *$\lambda : N \to \mathit{label}$ is a labelling function such that $x \not< y$ for all y whenever $\lambda(x) = \bot$.*
4. *$\varphi : N \to \mathit{form}$ is a formula function satisfying:*
   (a) *$\mathsf{sat}(\varphi(x))$ and $\mathsf{vars}(\varphi(x)) \subseteq {\downarrow}x$, for all $x \in N$;*
   (b) *$\varphi(y) \Rightarrow \varphi(x)$, for all $x < y$.*

LPOFs must also be binary branching, which means that for every $x \in N$ such that $\lambda(x) \in \mathit{etest}$, x has exactly two immediate successors y_1 and y_2 such that:

a. $\varphi(y_1) \Leftrightarrow \varphi(x) \wedge x$;
b. $\varphi(y_2) \Leftrightarrow \varphi(x) \wedge \neg x$; *and*
c. y_1 *and* y_2 *have no immediate predecessors other than* x.

Furthermore, if x is not the successor of a test, then $\varphi(x) \Leftrightarrow \bigwedge_{y<x} \varphi(y)$.

For some LPOF α, we use N_α, $<_\alpha$, λ_α, and φ_α to refer to its constituent parts. As is typical in pomset semantics, the partial order $\langle N, < \rangle$ denotes *causality*: $x < y$ iff x must be scheduled before y. If $x \not< y$ and $y \not< x$, then x and y execute concurrently and can be interleaved in any order. For example, see the diagrams below:



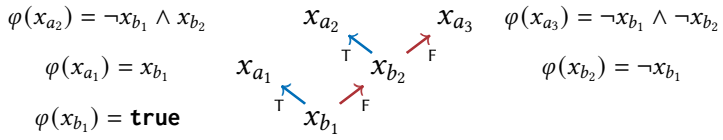
From left to right, the first LPOF is a sequential (totally ordered) program, where $x < y < z$. The second LPOF has a parallel branch, so that $x < y_1$ and $x < y_2$, but y_1 and y_2 have no relation, which means that y_1 and y_2 execute concurrently since one is not required to come before the other. The last LPOF also has a branch, but the branches eventually merge, as $y_1 < z$ and $y_2 < z$. Here, y_1 and y_2 still execute concurrently, but both must be executed before z.

With the examples in mind, conditions (2a) and (2b) are standard. Since each node represents an action, having finitely many predecessors ensures that every action may be scheduled in a finite amount of time. Moreover, requiring that every level has a finite number of nodes is a weakening of the usual finite branching property: a node can have infinitely many successors, but not all at the same level.

Concerning Condition (3), since $\bot$ denotes nontermination, nothing can occur after a $\bot$ node. Condition (4) is about formulae, which are crucial for modeling conditional constructs (if and

while), for example the program below corresponds to the LPOF structure underneath it.

$$\texttt{if } b_1 \texttt{ then } a_1 \texttt{ else if } b_2 \texttt{ then } a_2 \texttt{ else } a_3$$



(6.1)

Hence, every node x labelled with a Boolean test yields a (binary) branch and its (two) successors have the same formula as x, with an extra conjunct that is either x or $\neg x$, according to the outcome of the test. Thus, in Definition 6.2(4), requiring $\mathrm{sat}(\varphi(x))$ amounts to expressing the fact that every node x is reachable, *i.e.,* there exists a truth assignment v such that $v \vDash \varphi(x)$. The truth assignment v tells us whether each test passes or not. In our example, x_{a_3} is reachable when both the tests b_1 and b_2 fail, making $\varphi(x_{a_3})$ satisfiable.

That intuition about tests justifies the other two requirements of Definition 6.2(4): reachability of a node can only depend on the values of the tests that precede it (and so $\varphi(x)$ can only use nodes that precede x) and, since along a path we shall only add conjuncts, the formula of a higher-level node is stronger than the formulae of all of its predecessors.

6.2.2 Pomsets with Formulae

Although LPOFs express causality and branching behavior, they are not ideal semantic structures because information about node identifiers is extraneous. We instead work with *partially ordered multisets* (pomsets), which abstract away from the specific names. Pomsets are equivalence classes of LPOFs. To formalize equivalence classes, we first need to define isomorphism of LPOFs. Intuitively, two LPOFs are isomorphic if they are equal, modulo renaming the node identifiers.

Definition 6.3 (LPOF Isomorphism)**.** *Let α and β be LPOFs and $f : N_\alpha \to N_\beta$ be a bijection between their nodes. Define $f(\alpha) = \langle N, <, \lambda, \varphi \rangle$ as*

$$N \triangleq \{f(x) \mid x \in N_\alpha\} \qquad < \triangleq \{(f(x), f(y)) \mid x <_\alpha y\} \qquad \lambda \triangleq \lambda_\alpha \circ f^{-1} \qquad \varphi \triangleq f \circ \varphi \circ f^{-1}$$

where $f(\psi)$ syntactically renames the variables of ψ in the obvious way. We say that α and β are isomorphic, *written $\alpha \equiv \beta$, iff there is a bijection $f : N_\alpha \to N_\beta$ such that $f(\alpha) = \beta$.*

Pomsets with formulae are isomorphism classes of LPOFs.

Definition 6.4 (Pomsets with Formulae)**.** *Let $[\alpha] \triangleq \{\beta \in lpo \mid \alpha \equiv \beta\}$ be the isomorphism class of α. A* pomset with formulae $\boldsymbol{\alpha} \in pom$ *is an isomorphism class of LPOFs:*

$$pom \triangleq \{[\alpha] \mid \alpha \in lpo\}$$

6.2.3 Orders and Approximation

Next, we define an order relation, which describes what it means for one LPOF to *approximate* another. Intuitively, $\alpha \sqsubseteq_{lpo} \beta$ iff β has *more behaviors* than α, meaning that β can be obtained by expanding $\bot$ nodes in α into larger, more defined structures. For example, each unfolding of a while-loop produces a larger structure with respect to $\sqsubseteq_{lpo}$, as shown in Section 6.1.2.

Definition 6.5 (Ordering on LPOFs)**.** *We let $\alpha \sqsubseteq_{lpo} \beta$ iff:*

1. *N_α is a downward-closed subset of N_β,* i.e., *$N_\alpha \subseteq N_\beta$, and $x \in N_\beta$ if $y \in N_\beta$ and $x <_\alpha y$;*
2. *$<_\alpha = <_\beta \cap (N_\alpha \times N_\alpha)$;*
3. *For all $x \in N_\alpha$:*
   (a) *$\lambda_\alpha(x) \leq \lambda_\beta(x)$;*
   (b) *$\varphi_\alpha(x) = \varphi_\beta(x)$;*
   (c) *For all $y \in N_\beta$, either $y \in N_\alpha$ or $z <_\beta y$ for some $z \in N_\alpha$ such that $\lambda_\alpha(z) = \bot$.*

Definition 6.5 is a partial order, making $\langle lpo, \sqsubseteq_{lpo} \rangle$ a poset [Zilberstein et al. 2025a, Lemma C.3]. Moving to pomsets, we use the same order relation on the underlying LPOFs.

Definition 6.6 (Ordering on Pomsets)**.** *Let $\boldsymbol{\alpha} \sqsubseteq_{pom} \boldsymbol{\beta}$ iff $\exists \alpha \in \boldsymbol{\alpha}.\ \exists \beta \in \boldsymbol{\beta}.\ \alpha \sqsubseteq_{lpo} \beta$.*

We will now proceed to show that $\langle pom, \sqsubseteq_{pom} \rangle$ is a DCPO, and hence it is a suitable domain for defining fixed points. To do so, we must first show how to find suprema of directed sets of LPOFs, which is done by direct construction.

Lemma 6.7. *For any directed set $D \subseteq lpo$, $\sup D = \langle N, <, \lambda, \varphi \rangle$, where:*

$$N \triangleq \bigcup_{\beta \in D} N_\beta \qquad < \triangleq \bigcup_{\beta \in D} <_\beta \qquad \lambda(x) \triangleq \sup_{\beta \in D \,:\, x \in N_\beta} \lambda_\beta(x) \qquad \varphi(x) \triangleq \psi_x$$

where $\psi_x = \varphi_\beta(x)$, for all $\beta \in D$ such that $x \in N_\beta$, which is constant since D is directed.

Proof. The proof follows from Lemma 3.6 of Zilberstein et al. [2025a]. □

Stemming from the DCPO structure on LPOFs, we get that *pom* is also a DCPO.

Lemma 6.8. $\langle pom, \sqsubseteq_{pom} \rangle$ *is a pointed* DCPO.

Proof. $\langle pom, \sqsubseteq_{pom} \rangle$ is a DCPO by Lemma D.6 of Zilberstein et al. [2025a]. Clearly, the singleton pomset with label $\bot$ at the root is the bottom of the $\sqsubseteq_{pom}$ order. □

When manipulating pomsets, it is often convenient to work with finite structures. For example, linearization (Section 6.4) is most naturally defined recursively, but recursion is not well-founded for infinite structures, as the number of remaining nodes to process does not decrease each iteration. Instead, we can define linearization (and related operations) recursively on *finite approximations* of the infinite structure. To linearize the entire structure, we simply take the supremum of the linearization of the finite approximations. We now develop the necessary theory to justify why that approach is correct. First, we need a notion of a finite pomset. Let lpo_{fin} and pom_{fin} denote the set of LPOFs and pomsets with finitely many nodes, respectively.

$$lpo_{\text{fin}} \triangleq \{\alpha \in lpo \mid |N_\alpha| < \infty\} \qquad pom_{\text{fin}} \triangleq \{[\alpha] \mid \alpha \in lpo_{\text{fin}}\}$$

Given that, we write $\boldsymbol{\alpha} \ll \boldsymbol{\beta}$ to mean that $\boldsymbol{\alpha}$ is a finite approximation of $\boldsymbol{\beta}$, in other words:

$$\boldsymbol{\alpha} \ll \boldsymbol{\beta} \quad \text{iff} \quad \boldsymbol{\alpha} \in pom_{\text{fin}} \quad \text{and} \quad \boldsymbol{\beta} \in pom \quad \text{and} \quad \boldsymbol{\alpha} \sqsubseteq_{pom} \boldsymbol{\beta}$$

In order to define operations such as linearization on finite approximations, it is crucial that the aggregate of the finite approximations include *all* the information about the infinite structure, which is true since every node must appear at a finite level. Formally, any pomset $\boldsymbol{\alpha}$ coincides to the supremum of its finite approximations $\boldsymbol{\alpha} = \sup_{\boldsymbol{\beta} \ll \boldsymbol{\alpha}} \boldsymbol{\beta}$ [Zilberstein et al. 2025a, Lemma 3.9].

The final step is to show that functions defined on finite pomsets can be extended to functions on infinite pomsets in a well-behaved way. The extension of f is performed by taking the supremum of f applied to each finite approximation. For that supremum to be defined, f must be monotone, otherwise the set of results may not be directed, and hence it may not have a supremum. In that case, we prove that the extension is not only well-defined, but also Scott continuous.

Lemma 6.9 (Extension)**.** *Let $f\colon pom^n_{\text{fin}} \to T$ be a monotone function on the* DCPO *$\langle T, \leq\rangle$. Then $f^*\colon pom^n \to T$, shown below, is well-defined and Scott continuous:*

$$f^*(\boldsymbol{\alpha}_1, \ldots, \boldsymbol{\alpha}_n) \triangleq \sup_{\boldsymbol{\alpha}'_1 \ll \boldsymbol{\alpha}_1} \cdots \sup_{\boldsymbol{\alpha}'_n \ll \boldsymbol{\alpha}_n} f(\boldsymbol{\alpha}'_1, \ldots, \boldsymbol{\alpha}'_n)$$

Proof. The proof follows from Lemma 3.10 of Zilberstein et al. [2025a]. □

Scott continuity is important because it means that $f(\boldsymbol{\alpha}) = f^*(\boldsymbol{\alpha})$ for all $\boldsymbol{\alpha} \in pom_{\text{fin}}(L)$, *i.e.,* extension does not change the behavior of f when applied to finite pomsets.

6.3 POMSET SEMANTICS

We now define the denotational semantics for commands $C \in cmd_{\text{PC}}$ (see Figure 6.1) based on pomsets with fourmulae. The semantic function $[\![-]\!]_{pom} : cmd_{\text{PC}} \to pom$, mapping commands to pomsets, is shown in Figure 6.3 and relies on several new operations that will be defined throughout this section. We first define the singleton LPOF $\langle \ell \rangle_x$ on node x with label $\ell \in label$ and $\langle \ell \rangle$ on pomsets, which will be used to give the semantics to **`skip`** and atomic actions:

$$\begin{aligned} \langle \ell \rangle_x &\triangleq \langle \{x\}, \emptyset, [x \mapsto \ell], [x \mapsto \texttt{true}] \rangle && \in lpo \\ \langle \ell \rangle &\triangleq \{\langle \ell \rangle_x \mid x \in nodes\} && \in pom \end{aligned}$$

Singletons are the basic build block for pomsets; larger pomsets will be constructed by composing singletons. In the following sections, we will see three such operators for composing pomsets.

$$
\begin{aligned}
[\![\mathbf{skip}]\!]_{pom} &\triangleq \langle\, \bullet \,\rangle \\
[\![a]\!]_{pom} &\triangleq \langle a \rangle \\
[\![C_1 \fatsemi C_2]\!]_{pom} &\triangleq [\![C_1]\!]_{pom} \fatsemi [\![C_2]\!]_{pom} \\
[\![C_1 \parallel C_2]\!]_{pom} &\triangleq [\![C_1]\!]_{pom} \parallel [\![C_2]\!]_{pom} \\
[\![C_1 \,\&\, C_2]\!]_{pom} &\triangleq \mathsf{guard}(\&, [\![C_1]\!]_{pom}, [\![C_2]\!]_{pom}) \\
[\![C_1 \oplus_e C_2]\!]_{pom} &\triangleq \mathsf{guard}(\oplus_e, [\![C_1]\!]_{pom}, [\![C_2]\!]_{pom}) \\
[\![\mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2]\!]_{pom} &\triangleq \mathsf{guard}(b, [\![C_1]\!]_{pom}, [\![C_2]\!]_{pom}) \\
[\![\mathbf{while}\ b\ \mathbf{do}\ C]\!]_{pom} &\triangleq \mathsf{lfp}\left(\Psi_{\langle C,b\rangle}\right) \\
\text{where } \Psi_{\langle C,b\rangle}(\boldsymbol{\alpha}) &\triangleq \mathsf{guard}(b, [\![C]\!]_{pom} \fatsemi \boldsymbol{\alpha}, \langle\, \bullet \,\rangle)
\end{aligned}
$$

Figure 6.3: Pomset semantics of commands $[\![-]\!]_{pom} : \mathit{cmd}_{\mathsf{PC}} \to \mathit{pom}$.

6.3.1 Guarded Choice

As we saw in Section 6.1, the key novel functionality of pomsets with formulae is the ability to encode guarded branching using Boolean formulae to express the sequence of tests that must pass or fail to reach each point in the structure. We now define the operation:

$$\mathsf{guard} \colon \mathit{nodes} \times \mathit{etest} \times \mathit{lpo} \times \mathit{lpo} \to \mathit{lpo}$$

which constructs an LPOF with a test at the root, and two branches corresponding to the cases where the test passes and fails, respectively. Assuming that $N_\alpha \cap N_\beta = \emptyset$ and $x \notin N_\alpha \cup N_\beta$, we let $\mathsf{guard}(x, t, \alpha, \beta) = \langle N, <, \lambda, \varphi \rangle$ where we overload $\wedge$ s.t. $\psi \wedge \mathbf{true} = \mathbf{true} \wedge \psi = \psi$, and:

$$
N \triangleq \{x\} \cup N_\alpha \cup N_\beta \qquad\qquad < \triangleq <_\alpha \cup <_\beta \cup \left(\{x\} \times (N_\alpha \cup N_\beta)\right)
$$

$$
\lambda(y) \triangleq \begin{cases} t & \text{if } y = x \\ \lambda_\alpha(y) & \text{if } y \in N_\alpha \\ \lambda_\beta(y) & \text{if } y \in N_\beta \end{cases} \qquad\qquad \varphi(y) \triangleq \begin{cases} \mathbf{true} & \text{if } y = x \\ \varphi_\alpha(y) \wedge x & \text{if } y \in N_\alpha \\ \varphi_\beta(y) \wedge \neg x & \text{if } y \in N_\beta \end{cases}
$$

So, $\mathsf{guard}(x, b, \alpha, \beta)$ joins α and β with a new root node x, whose label is t, and additionally, the formulae in α and β are updated to require that t passes or fails, respectively. As an example, $\mathsf{guard}(x_{b_1}, b_1, \langle a_1 \rangle_{x_{a_1}}, \mathsf{guard}(x_{b_2}, b_2, \langle a_2 \rangle_{x_{a_2}}, \langle a_3 \rangle_{x_{a_3}}))$ produces the LPOF depicted in (6.1). For finite

pomsets, we define guard: $\mathit{etest} \times \mathit{pom}_{\mathsf{fin}} \times \mathit{pom}_{\mathsf{fin}} \to \mathit{pom}_{\mathsf{fin}}$ simply as the equivalence class of the corresponding LPOFS:

$$\mathrm{guard}(t, \boldsymbol{\alpha}, \boldsymbol{\beta}) \triangleq \left\{ \mathrm{guard}(x, t, \alpha, \beta) \;\middle|\; \alpha \in \boldsymbol{\alpha},\ \beta \in \boldsymbol{\beta}, N_\alpha \cap N_\beta = \emptyset, x \notin N_\alpha \cup N_\beta \right\}$$

Unsurprisingly, guard is monotone, as making either branch larger will make the composed structure larger too. Therefore, we define guard: $\mathit{etest} \times \mathit{pom} \times \mathit{pom} \to \mathit{pom}$ for infinite structures by extension, and we get that it is Scott Continuous from Lemma 6.9.

$$\mathrm{guard}(\ell, \boldsymbol{\alpha}, \boldsymbol{\beta}) \quad \triangleq \quad \mathrm{guard}^*(\ell, \boldsymbol{\alpha}, \boldsymbol{\beta}) \quad = \quad \sup_{\boldsymbol{\alpha}' \ll \boldsymbol{\alpha}} \sup_{\boldsymbol{\beta}' \ll \boldsymbol{\beta}} \mathrm{guard}(\ell, \boldsymbol{\alpha}', \boldsymbol{\beta}')$$

The semantics of if statements, nondeterministic branching, and probabilistic branching are all defined using guard, but with different kinds of extended tests.

$$[\![\texttt{if}\ b\ \texttt{then}\ a_1\ \texttt{else}\ a_2]\!]_{pom} = \begin{array}{ccc} a_1 & & a_2 \\ {\scriptstyle \mathsf{T}}\nwarrow & & \nearrow{\scriptstyle \mathsf{F}} \\ & b & \end{array}$$

$$[\![a_1 \,\&\, a_2]\!]_{pom} = \begin{array}{ccc} a_1 & & a_2 \\ {\scriptstyle \mathsf{T}}\nwarrow & & \nearrow{\scriptstyle \mathsf{F}} \\ & \& & \end{array} \qquad\qquad [\![a_1 \oplus_e a_2]\!]_{pom} = \begin{array}{ccc} a_1 & & a_2 \\ {\scriptstyle \mathsf{T}}\nwarrow & & \nearrow{\scriptstyle \mathsf{F}} \\ & \oplus_e & \end{array}$$

6.3.2 Sequential Composition

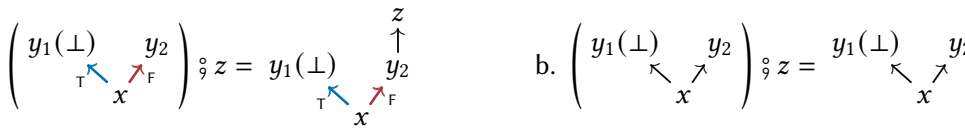
The sequential composition operation $\boldsymbol{\alpha} \fatsemi \boldsymbol{\beta}$ enforces that all actions in $\boldsymbol{\alpha}$ must occur before any of those in $\boldsymbol{\beta}$. Due to the interaction between causality and guarded branching, the formal definition of $\fatsemi$ is complex. To see the complexity, consider the following example LPOFS (where x denotes the singleton $\langle \ell \rangle_x$ when the label is not relevant):

$$\text{a. } x \fatsemi y = \begin{array}{c} y \\ \uparrow \\ x \end{array} \qquad \text{b. } \begin{pmatrix} y_1 & & y_2 \\ & \nwarrow\ \nearrow & \\ & x & \end{pmatrix} \fatsemi z = \begin{array}{ccc} & z & \\ \nearrow & & \nwarrow \\ y_1 & & y_2 \\ & \nwarrow\ \nearrow & \\ & x & \end{array} \qquad \text{c. } \begin{pmatrix} y_1 & & y_2 \\ {\scriptstyle \mathsf{T}}\nwarrow & & \nearrow{\scriptstyle \mathsf{F}} \\ & x & \end{pmatrix} \fatsemi z = \begin{array}{ccc} z_x & & z_{\neg x} \\ \uparrow & & \uparrow \\ y_1 & & y_2 \\ {\scriptstyle \mathsf{T}}\nwarrow & & \nearrow{\scriptstyle \mathsf{F}} \\ & x & \end{array} \tag{6.2}$$

The sequential composition of two singletons (6.2a) simply creates a causality between them. When the program has forked into parallel (unguarded) branches (6.2b), $\fatsemi$ introduces a diamond structure, as is typical in pomset semantics. However, if the program has a *guarded* branch (6.2c), then the following actions must be copied after each branch. Copying is essential to correctly

account for loops: in the program (**while** b **do** a) ; a', if we only created a single node labelled with a', then that node would not be finitely preceded.

The definition of $\fatsemi$ must also not introduce actions to occur after a $\bot$ node. Consider the following LPOFSs, where node y_1 is labelled with $\bot$ and all other nodes have non-$\bot$ labels:



(6.3)

When composing the singleton z after a guarded branch with one stuck path (6.3a), z is only added to the non-stuck branch. In the program execution, only one branch will be taken, so z can be safely executed as long as the test at x is false. However, the same is not true for parallel branching: in (6.3b), both y_1 and y_2 will eventually be scheduled, so z must occur after y_1—which is equivalent to not occurring at all—thus it does not appear in the final structure. Dealing properly with $\bot$ ensures that $\fatsemi$ is monotonic.

Building on the above intuitions, we first define $\fatsemi$ for finite LPOFSs and pomsets, and then extend our definition to infinite ones using Lemma 6.9. In the rest of this section, we assume that $\alpha, \beta \in \mathit{lpo}_{\mathsf{fin}}(L)$. We start by defining the notion of stuck computations, extensible nodes, and branches. The formula stuck_α indicates which nodes are guaranteed to encounter a $\bot$ later in their execution. Extensible nodes $x \in \mathsf{ext}_\alpha$ are not stuck, so there is some computation path that they can take without encountering any $\bot$ node.

$$\mathsf{stuck}_\alpha \triangleq \bigvee_{x \in N_\alpha : \lambda_\alpha(x) = \bot} \varphi_\alpha(x) \qquad\qquad \mathsf{ext}_\alpha \triangleq \{x \in N_\alpha \mid \varphi_\alpha(x) \not\Rightarrow \mathsf{stuck}_\alpha\}$$

In (6.2), all nodes are extensible, whereas in (6.3a) only x and y_2 are extensible and in (6.3b) none are extensible. The set of branches br_α contains all of the maximal formulae that can be obtained as the conjunction of formulae of extensible nodes:

$$\mathsf{br}_\alpha \triangleq \{\varphi_\alpha(S) \mid \emptyset \subset S \subseteq \mathsf{ext}_\alpha,\ \varphi_\alpha(S) \Rightarrow \neg\mathsf{stuck}_\alpha,\ \forall T.\ S \subset T \subseteq \mathsf{ext}_\alpha \Rightarrow \neg\mathsf{sat}(\varphi_\alpha(T))\}$$

where, $\varphi_\alpha(S) \triangleq \bigwedge_{x \in S} \varphi_\alpha(x)$ for any $S \subseteq N_\alpha$. For example, in (6.2a,b), $\mathsf{br} = \{\mathbf{true}\}$ since there are no tests; in (6.2c), $\mathsf{br} = \{x, \neg x\}$ since both outcomes of the test are not stuck; in (6.3a), $\mathsf{br} = \{\neg x\}$ since the program is stuck if x passes; and in (6.3b), $\mathsf{br} = \emptyset$, since all paths are stuck.

As we saw in (6.2c), we will need an isomorphic copy $\beta_\psi \equiv \beta$ for each branch $\psi \in \mathrm{br}_\alpha$. Those copies are generated by a function $f\colon \mathrm{br}_\alpha \to [\beta]$ such that the nodes of $f(\psi)$ are disjoint from those of α and of every $f(\psi')$ where $\psi \neq \psi'$. The function f is drawn from the following set:

$$\mathrm{copy}_{\alpha,\beta} \triangleq \left\{ f\colon \mathrm{br}_\alpha \to [\beta] \;\middle|\; \forall \psi \in \mathrm{br}_\alpha.\, N_{f(\psi)} \cap N_\alpha = \emptyset \;\wedge\; \forall \psi' \neq \psi.\, N_{f(\psi)} \cap N_{f(\psi')} = \emptyset \right\}$$

Given $\alpha, \beta \in \mathit{lpo}_{\mathsf{fin}}(L)$ and $f \in \mathrm{copy}_{\alpha,\beta}$, we define sequential composition $\alpha \fatsemi_f \beta = \langle N, <, \lambda, \varphi \rangle$ where the components are defined as follows (notationally, we let $\beta_\psi \triangleq f(\psi)$):

$$N \triangleq N_\alpha \cup \bigcup_{\psi \in \mathrm{br}_\alpha} N_{\beta_\psi} \qquad < \triangleq <_\alpha \cup \bigcup_{\psi \in \mathrm{br}_\alpha} \left(<_{\beta_\psi} \cup (\{x \in N_\alpha \mid \psi \Rightarrow \varphi_\alpha(x)\} \times N_{\beta_\psi}) \right)$$

$$\lambda(x) \triangleq \begin{cases} \lambda_\alpha(x) & \text{if } x \in N_\alpha \\ \lambda_{\beta_\psi}(x) & \text{if } x \in N_{\beta_\psi} \end{cases} \qquad \varphi(x) \triangleq \begin{cases} \varphi_\alpha(x) & \text{if } x \in N_\alpha \\ \varphi_{\beta_\psi}(x) \wedge \psi & \text{if } x \in N_{\beta_\psi} \end{cases}$$

(where, again, $\psi \wedge \mathbf{true} = \mathbf{true} \wedge \psi = \psi$). So, the nodes of $\alpha \fatsemi_f \beta$ are the nodes of α, and all the nodes of the isomorphic copies β_ψ. The new order preserves the causalities in α and in each β_ψ, and additionally requires that β_ψ occurs after all the nodes in the branch ψ of $\boldsymbol{\alpha}$. All labels are preserved, and formulae in β_ψ are updated to also include ψ. Sequential composition for finite pomsets is defined by taking the equivalence class of all underlying LPOFS:

$$\boldsymbol{\alpha} \fatsemi \boldsymbol{\beta} \triangleq \{\alpha \fatsemi_f \beta \mid \alpha \in \boldsymbol{\alpha},\ \beta \in \boldsymbol{\beta},\ f \in \mathrm{copy}_{\alpha,\beta}\}$$

Following from the monotonicity properties of stuck, extensible nodes, and branches, sequential composition is also monotone, therefore sequential composition of infinite pomsets is defined by extension, and is Scott continuous:

$$\boldsymbol{\alpha} \fatsemi \boldsymbol{\beta} \quad \triangleq \quad \boldsymbol{\alpha} \fatsemi^* \boldsymbol{\beta} \quad = \quad \sup_{\boldsymbol{\alpha}' \ll \boldsymbol{\alpha}} \sup_{\boldsymbol{\beta}' \ll \boldsymbol{\beta}} \boldsymbol{\alpha}' \fatsemi \boldsymbol{\beta}'$$

6.3.3 Parallel Composition

We finally define parallel composition. We reuse the no-op label $\bullet$ to denote thread forking and let $\mathsf{nFork}_\alpha \triangleq N_\alpha \setminus \{x \in N_\alpha \mid \lambda_\alpha(x) = \bullet \wedge \forall y \in N_\alpha \setminus \{x\}.\ x <_\alpha y\}$ denote the set of nodes of α with the root removed if the root is a $\bullet$ node. Then, for both finite and infinite LPOFs α and β, and for any $x \notin N_\alpha \cup N_\beta$, we let $\alpha \parallel_x \beta \triangleq \langle N, <, \lambda, \varphi \rangle$, where

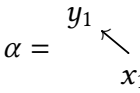
$$N \triangleq \{x\} \cup \mathsf{nFork}_\alpha \cup \mathsf{nFork}_\beta$$

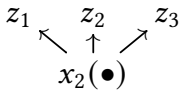
$$< \triangleq (\{x\} \times (\mathsf{nFork}_\alpha \cup \mathsf{nFork}_\beta)) \cup (<_\alpha \cap (\mathsf{nFork}_\alpha \times \mathsf{nFork}_\alpha)) \cup (<_\beta \cap (\mathsf{nFork}_\beta \times \mathsf{nFork}_\beta))$$

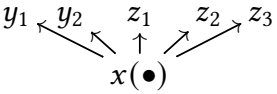
$$\lambda(y) \triangleq \begin{cases} \bullet & \text{if } y = x \\ \lambda_\alpha(y) & \text{if } y \in \mathsf{nFork}_\alpha \\ \lambda_\beta(y) & \text{if } y \in \mathsf{nFork}_\beta \end{cases} \qquad \varphi(y) \triangleq \begin{cases} \mathtt{true} & \text{if } y = x \\ \varphi_\alpha(y) & \text{if } y \in \mathsf{nFork}_\alpha \\ \varphi_\beta(y) & \text{if } y \in \mathsf{nFork}_\beta \end{cases}$$

Notice that the branching that arises from $\parallel$ is conceptually (and practically) different from the branching that arises from a test: $\parallel$ does not exclude either branch, whereas the two branches due to a test are mutually exclusive. The difference is apparent in the definition of the formulae associated to the nodes after the branch, which are not extended with a new conjunct under parallel composition. The complication arising in our handling of parallel composition through $\bullet$ is that we want one single node labelled with $\bullet$, even if we put in parallel many LPOFs.

For example, by putting in parallel the singleton LPOFs made up, respectively, by nodes y_1 and y_2 and by nodes z_1, z_2 and z_3 (and where $\bullet$ is the label of the root nodes, say x_1 and x_2 respectively), we obtain the LPOFs α, β, and their parallel composition as follows:

$\alpha =$ 

$\beta =$ 

$\alpha \parallel_x \beta =$ 

where x_1 and x_2 disappear and the new root is x, labelled $\bullet$.

As a side note, observe that the way in which we defined parallel composition entails that $\alpha \sqsubseteq_{lpo} \alpha \parallel_x \beta$ if and only if $\alpha = \langle \bot \rangle_x$: the LPOF α has just a node x labelled with $\bot$; the second LPOF has node x labelled with $\bullet$ that precedes a node labelled with $\bot$ and all the nodes of β. Also, a somewhat unusual notion of associativity for $\parallel$ holds: $(\alpha \parallel_x \beta) \parallel_y \gamma = \alpha \parallel_y (\beta \parallel_z \gamma)$, but the

usual associativity is recovered for pomsets, where we abstract from the specific node set[1]. Like before, we define parallel composition for (finite and infinite) pomsets on top of that for LPOFS:

$$\boldsymbol{\alpha} \parallel \boldsymbol{\beta} \triangleq \{\alpha \parallel_x \beta \mid \alpha \in \boldsymbol{\alpha},\ \beta \in \boldsymbol{\beta},\ x \notin N_\alpha \cup N_\beta\}$$

6.3.4 Fixed Points and Loops

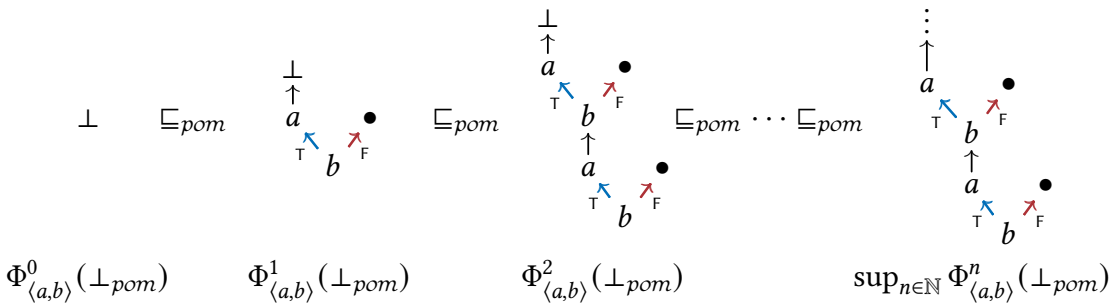
While loops are interpreted as the least fixed point of the function $\Psi_{\langle C,b\rangle} : pom \to pom$.

$$\Psi_{\langle C,b\rangle}(\boldsymbol{\alpha}) \triangleq \mathrm{guard}(b, [\![C]\!]_{pom} \fatsemi \boldsymbol{\alpha}, \langle \bullet \rangle)$$

Clearly $\Psi_{\langle C,b\rangle}$ is Scott continuous, as it is defined using the Scott continuous operations guard and $\fatsemi$. So, by Kleene's fixed point theorem, $\Psi_{\langle C,b\rangle}$ has a least fixed point given by:

$$[\![\textbf{while } b \textbf{ do } C]\!]_{pom} \quad = \quad \mathrm{lfp}\left(\Psi_{\langle C,b\rangle}\right) \quad = \quad \sup_{n\in\mathbb{N}} \Psi^n_{\langle C,b\rangle}\left(\bot_{pom}\right)$$

Considering a loop $[\![\textbf{while } b \textbf{ do } a]\!]_{pom}$ with a single action a as its body, we can unroll the definition several times to get the following infinite chain:



The supremum of the chain above is clearly an infinite structure with a terminating branch for each $n \in \mathbb{N}$, and an infinitely ascending spine. The $\bot$ node is pushed to a progressively higher level after each unrolling, so in the supremum it does not appear at all.

[1] We remark that $\fatsemi$ on LPOFS also satisfies a non-straightforward associativity, that however becomes the usual one when passing to pomsets.

$$\mathcal{L}_{lpo}(\alpha,\psi,S)(\sigma) \triangleq \begin{cases} \eta(s) & \text{if } \mathsf{next}(\alpha,\psi,S) = \emptyset \\ \&_{x\in\mathsf{next}(\alpha,\psi,S)}\, \mathcal{L}_{node}(\alpha,\psi,S,x)(\sigma) & \text{if } \mathsf{next}(\alpha,\psi,S) \neq \emptyset \end{cases}$$

$$\mathcal{L}_{node}(\alpha,\psi,S,x)(\sigma) \triangleq \begin{cases} \mathcal{L}_{lpo}(\alpha,\psi,S\cup\{x\})^{\dagger}\left(\llbracket\lambda_\alpha(x)\rrbracket_{act}\,(\sigma)\right) & \text{if } \lambda_\alpha(x) \in act \\ \mathcal{L}_{test}(\alpha,\psi,S,x)(\sigma)^{\dagger}(\llbracket\lambda_\alpha(x)\rrbracket_{etest}\,(\sigma)) & \text{if } \lambda_\alpha(x) \in etest \\ \bot_C & \text{if } \lambda_\alpha(x) = \bot \\ \mathcal{L}_{lpo}(\alpha,\psi,S\cup\{x\})(\sigma) & \text{if } \lambda_\alpha(x) = \bullet \end{cases}$$

$$\mathcal{L}_{test}(\alpha,\psi,S,x)(\sigma)(b) \triangleq \begin{cases} \mathcal{L}_{lpo}(\alpha,\psi \wedge \quad x, S\cup\{x\})(\sigma) & \text{if } b = 1 \\ \mathcal{L}_{lpo}(\alpha,\psi \wedge \neg x, S\cup\{x\})(\sigma) & \text{if } b = 0 \end{cases}$$

$$\mathsf{next}(\alpha,\psi,S) \triangleq \{x \in N_\alpha \setminus S \mid {\downarrow}x \subseteq S \wedge \psi \Rightarrow \varphi_\alpha(x)\}$$

$$\llbracket t\rrbracket_{etest}\,(\sigma) \triangleq \begin{cases} \eta(\llbracket t\rrbracket_{test}\,(\sigma)) & \text{if } t \in test \\ \eta(1)\,\&\,\eta(0) & \text{if } t = \& \\ \eta(1) \oplus_{\llbracket e\rrbracket_{exp}(\sigma)} \eta(0) & \text{if } t = \oplus_e \end{cases}$$

Figure 6.4: Linearization for finite LPOFS $\mathcal{L}_{lpo} : lpo_{\mathsf{fin}} \times form \times 2^{nodes} \to \mathsf{Mem}[V] \to C(\mathsf{Mem}[V])$.

6.4 LINEARIZATION AND ADEQUACY

We now discuss how to produce a state transformer from the pomset semantics of Section 6.3, which is useful for understanding the input-output behavior of programs resulting from scheduling concurrent threads. The state transformer is obtained via *linearization*, which considers all interleavings of the threads in order to interpret the semantics of a *complete* program as a function from inputs into the convex powerdomain. Once the program is linearized, more threads cannot be composed in parallel, as the causality information is gone.

We start by defining a linearization operation on finite LPOFS, which is shown in Figure 6.4. Linearization is defined recursively, until there are no more nodes to schedule. It must be defined on finite structures, otherwise the recursion is not well-founded, but it can be extended to infinite structures using the extension lemma. In $\mathcal{L}_{lpo}(\alpha,\psi,S)$, the set $S \subseteq N_\alpha$ contains all the nodes that have already been processed and ψ is a path condition, indicating the outcomes of the tests

associated to the nodes in S.

The function $\mathsf{next}(\alpha, \psi, S)$ gives the nodes that are ready to be scheduled, which includes all nodes that obey the path condition ψ, and whose predecessors are already in S. If $\mathsf{next}(\alpha, \psi, S)$ is empty, then $\mathcal{L}_{lpo}(\alpha, \psi, S)$ simply returns the current state s using the monad unit η, since there are no more nodes to schedule. If not, then it nondeterministically selects a next node using $\&$. If the next node is an action, then it interprets the action using $[\![-]\!]_{act}$ and uses Kleisli composition to compose the result with the linearization of the remaining LPOF. If the next node is a test, then that test is evaluated and the path condition is updated to account for the outcome of the test. Standard tests are evaluated using $[\![-]\!]_{test}$ from Section 2.1.3, whereas $\&$ and $\oplus_e$ return 0 or 1 either nondeterministically and probabilistically, respectively. If the next node is $\bot$, then the linearization is $\bot_C$, and $\bullet$ is treated like a no-op.

We next define linearization on finite pomsets $\mathcal{L}_{\mathsf{fin}} : pom_{\mathsf{fin}} \to \mathsf{Mem}[V] \to C(\mathsf{Mem}[V])$ using $\mathcal{L}_{lpo}$. Clearly $\mathcal{L}_{lpo}(\alpha, \mathbf{true}, \emptyset) = \mathcal{L}_{lpo}(\beta, \mathbf{true}, \emptyset)$ if $\alpha \equiv \beta$, so $\mathcal{L}_{\mathsf{fin}}(\boldsymbol{\alpha})$ can be defined for any arbitrary representative LPOF $\alpha \in \boldsymbol{\alpha}$ using the trivial path condition $\mathbf{true}$ and an empty set of visited nodes:

$$\mathcal{L}_{\mathsf{fin}}([\alpha]) \triangleq \mathcal{L}_{lpo}(\alpha, \mathbf{true}, \emptyset)$$

Linearization of finite pomsets is monotone [Zilberstein et al. 2025a, Lemma G.3], therefore, linearization can be extended to infinite pomsets $\mathcal{L} : pom \to \mathsf{Mem}[V] \to C(\mathsf{Mem}[V])$ by extension, which is well-defined and Scott continuous.

$$\mathcal{L}(\boldsymbol{\alpha}) \quad \triangleq \quad \mathcal{L}^*_{\mathsf{fin}}(\boldsymbol{\alpha}) \quad = \quad \sup_{\boldsymbol{\alpha}' \ll \boldsymbol{\alpha}} \mathcal{L}_{\mathsf{fin}}(\boldsymbol{\alpha}')$$

To ensure that linearization acts as desired, we provide a sanity check lemma to show that the linearized semantics behaves in the standard way from Section 2.1. Linearizing $\mathbf{skip}$ gives the monad unit; linearizing a singleton action has the same behavior as interpreting the action; linearizing a sequential composition is equal to Kleisli composition of the individual linearizations; linearizing an if-statement is equal to linearizing one of the two branches, depending on the truth

of the guard; and the linearization of a while loop is equal to the least fixed point over a different characteristic function $\Xi_{\langle C,b\rangle}$. The function $\Xi_{\langle C,b\rangle}$ inherits Scott continuity from the Kleisli composition operator, therefore the fixed point exists.

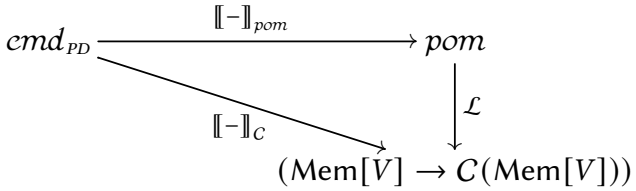
Lemma 6.10 (Linearization). *The following properties hold:*

$$\begin{aligned}
\mathcal{L}(\llbracket \mathbf{skip} \rrbracket_{pom}) &= \eta \\
\mathcal{L}(\llbracket a \rrbracket_{pom}) &= \llbracket a \rrbracket_{act} \\
\mathcal{L}(\llbracket C_1 \fatsemi C_2 \rrbracket_{pom}) &= \mathcal{L}(\llbracket C_2 \rrbracket_{pom})^\dagger \circ \mathcal{L}(\llbracket C_1 \rrbracket_{pom}) \\
\mathcal{L}(\llbracket C_1 \,\&\, C_2 \rrbracket_{pom})(\sigma) &= \mathcal{L}(\llbracket C_1 \rrbracket_{pom})(\sigma) \,\&\, \mathcal{L}(\llbracket C_2 \rrbracket_{pom})(\sigma) \\
\mathcal{L}(\llbracket C_1 \oplus_e C_2 \rrbracket_{pom})(\sigma) &= \mathcal{L}(\llbracket C_1 \rrbracket_{pom})(\sigma) \oplus_{\llbracket e \rrbracket_{exp}(\sigma)} \mathcal{L}(\llbracket C_2 \rrbracket_{pom})(\sigma) \\
\mathcal{L}(\llbracket \mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2 \rrbracket_{pom})(\sigma) &= \begin{cases} \mathcal{L}(\llbracket C_1 \rrbracket_{pom})(\sigma) & \text{if } \llbracket b \rrbracket_{test}(\sigma) = 1 \\ \mathcal{L}(\llbracket C_2 \rrbracket_{pom})(\sigma) & \text{if } \llbracket b \rrbracket_{test}(\sigma) = 0 \end{cases} \\
\mathcal{L}\left(\llbracket \mathbf{while}\ b\ \mathbf{do}\ C \rrbracket_{pom}\right) &= \mathsf{lfp}\left(\Xi_{\langle C,b\rangle}\right) \\
\text{where } \Xi_{\langle C,b\rangle}(f)(\sigma) &\triangleq \begin{cases} f^\dagger\left(\mathcal{L}(\llbracket C \rrbracket_{pom}))(\sigma)\right) & \text{if } \llbracket b \rrbracket_{test}(\sigma) = 1 \\ \eta(\sigma) & \text{if } \llbracket b \rrbracket_{test}(\sigma) = 0 \end{cases}
\end{aligned}$$

Proof. Follows from Lemma 5.2 of Zilberstein et al. [2025a]. □

Lemma 6.10 is quite significant; it implies that $\mathcal{L}(\llbracket C \rrbracket_{pom})$ corresponds to the convex powerdomain semantics from Chapter 5.

Theorem 6.11 (Adequacy). *The following diagram commutes:*



In other words, $\mathcal{L}(\llbracket C \rrbracket_{pom}) = \llbracket C \rrbracket_C$ *for all* PDCL *program* $C \in cmd_{PD}$.

Proof. By induction on the structure of C. The cases follow immediately from Lemma 6.10. □

As a consequence of Theorem 6.11, the DOL inference rules carry over to a more complex logic for analyzing PCCL programs. But properties about parallel composition are notably missing from Lemma 6.10. In fact, no such compositional properties exist, since parallel composition can completely change the behavior of each thread, as we saw in Section 6.1. In the next chapter, we will develop compositional reasoning strategies for parallel programs based on separation logic.

6.5 RELATED WORK

Pomsets with formulae is a new semantic structure, which we use as basis to provide semantics for a probabilistic concurrent imperative language with unbounded loops. Our work builds on a rich history of using partial orders and pomsets in denotational models of concurrent programs [Grabowski 1981; Pratt 1976; Gischer 1988], which have been used as a semantic basis for concurrent separation logic [Brookes 2004], concurrent Kleene algebras [Hoare et al. 2011; Kappé et al. 2020; Laurence and Struth 2014; Jipsen and Moshier 2016], and other applications [Kavanagh and Brookes 2018; Edixhoven et al. 2022; Jagadeesan et al. 2020].

While there are links between pomset semantics and operational models of concurrency [Brookes 2002], pomsets are often preferred for their finer notion of concurrency compared to the interleaving semantics offered by operational models, which is especially useful in the context of weak memory [Kavanagh and Brookes 2018; Jagadeesan et al. 2020]. Probabilistic event structures model probabilistic process algebras [Katoen 1996; Winskel 2014; Varacca et al. 2006; Varacca and Yoshida 2007], whereas we model a full imperative language with sequential composition, control flow, and unbounded loops, for which adding formulae was essential.

Concurrency has previously been combined with other effects in limited ways. For example, semantics for probabilistic concurrent programs have been defined operationally [Tassarotti and Harper 2019; Tassarotti 2018], but that approach is limited to bounded loops due to the lack of domain-theoretic tools to determine the infinitary behavior. Whereas finite-trace models are sufficient for many non-probabilistic scenarios, unbounded looping in probabilistic programs re-

quires infinite traces, as the probability of termination may only become 1 in the limit. A proper theory of infinite traces requires pomsets to be enriched with a DCPO [Meyer and de Vink 1988, 1989] or metric [de Bakker and Warmerdam 1990] structure; prior work was very informative to our own development, although the inclusion of deterministic branching in our own pomset structure added significant complexity.

Fesefeldt et al. [2022] introduced an operational model of probabilistic concurrency based on Markov Decision Processes, which they used to lower bound expected values, but that approach does not straightforwardly extend to more complex domains such as the convex powerdomain, which gives the full set of distributions over outcomes.

Neves [2025] developed a denotational model where the underlying semantic structure is obtained as the solution to a domain equation. Our construction is more concrete, giving a full account of the causality in the program and thus captures non-interleaving models of concurrency too. In addition, our extension theorem provides a convenient basis for inductive proofs over unbounded programs, as we saw in Theorem 6.11 and will see again in Lemma 7.8.

Branching pomsets were recently introduced to model choices in choreographic programs [Edixhoven et al. 2022; Edixhoven and Jongmans 2022; Edixhoven et al. 2024]. While branching pomsets can, in theory, contain infinitely many nodes, the domain theoretic properties needed to approximate infinite structures have not been explored, which was a significant focus of this paper. Indeed, our extension lemma (Lemma 6.9) was necessary for linearization, without which we would not have been able to relate our pomeset model to the known convex powerset semantics [He et al. 1997; McIver and Morgan 2005; Zilberstein et al. 2025b].

CHAPTER 7

PROBABILISTIC CONCURRENT OUTCOME LOGIC

As a culmination of all the prior chapters, we now present Probabilistic Concurrent Outcome Logic (pcOL), which combines pOL's ability to reason about probabilistic outcomes and dOL's ability to reason about nondeterminism with compositional concurrency reasoning inspired by Concurrent Separation Logic (CSL) and semantics based on pomsets with formulae.

Throughout this chapter, we will see how invariant-sensitive semantics allows us to compositionally model parallel threads, even if they interact with shared state. Furthermore, the probabilistic independence guarantee of the separating conjunction allows us to compositionally reason about parallel threads, but still recover the joint distribution over all of the program variables. Observations about shared state can result in nondeterminism, so just like dOL, pcOL must carefully control for correlations between the parallel threads that might disrupt probabilistic independence guarantees. In the next chapter, we will see larger case studies where the capabilities that we have built up from pOL to dOL to pcOL are all on display.

7.1 OVERVIEW

As we saw in Section 2.4.3, Concurrent Separation Logic (CSL) achieves compositionality via *separation* [O'Hearn 2004; Brookes 2004]. The Par rule states that if two threads act on disjoint memory regions, then their behavior will not change when they are run in parallel.

$$\frac{\{P_1\}\ C_1\ \{Q_1\} \qquad \{P_2\}\ C_2\ \{Q_2\}}{\{P_1 * P_2\}\ C_1 \parallel C_2\ \{Q_1 * Q_2\}}\textsc{Par}$$

The central question behind pcOL is how to bring the kind of compositional concurrency reasoning that was pioneered by CSL into a more expressive setting where we can reason about probabilistic and nondeterministic outcomes, as we did in pOL and dOL.

7.1.1 Concurrent Separation Logic meets Demonic Outcome Logic

From pOL and dOL, we already know that if we separately flip two coins and store the results in the variables x and y, then the respective variables will be distributed according to Bernoulli distributions with parameter $\frac{1}{2}$.

$$\langle \lceil x \mapsto - \rceil \rangle\ x :\approx \mathbf{Ber}\left(\tfrac{1}{2}\right)\ \langle x \sim \mathbf{Ber}\left(\tfrac{1}{2}\right) \rangle \qquad \langle \lceil y \mapsto - \rceil \rangle\ y :\approx \mathbf{Ber}\left(\tfrac{1}{2}\right)\ \langle y \sim \mathbf{Ber}\left(\tfrac{1}{2}\right) \rangle$$

As we have seen repeatedly throughout the prior chapters, probabilistic independence is a powerful tool for compositionally determining the joint distribution over the local and global memories in a program. Indeed, pOL and dOL provide compositional ways to combine the two specifications above. For example, if they were composed in a sequence, then we could use Frame and Seq to derive the following specification.

$$\langle \lceil x \mapsto - \rceil * \lceil y \mapsto - \rceil \rangle\ x :\approx \mathbf{Ber}\left(\tfrac{1}{2}\right) \fatsemi y :\approx \mathbf{Ber}\left(\tfrac{1}{2}\right)\ \langle x \sim \mathbf{Ber}\left(\tfrac{1}{2}\right) * y \sim \mathbf{Ber}\left(\tfrac{1}{2}\right) \rangle$$

Independence allows us to recover the entire joint distribution over all the program variables above, for example, we know that x and y are both 1 with probability $\frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4}$.

Going forward, a version of Par based on probabilistic separation would give us a powerful way to compositionally reason about probabilistic concurrency, which fully dictates the joint distribution over all of the program variables. Instead of composing the two coin flips in sequence, we could compose them in parallel and obtain the specification below.

$$\langle \lceil x \mapsto - \rceil * \lceil y \mapsto - \rceil \rangle\ x :\approx \mathbf{Ber}\left(\tfrac{1}{2}\right) \parallel y :\approx \mathbf{Ber}\left(\tfrac{1}{2}\right)\ \langle x \sim \mathbf{Ber}\left(\tfrac{1}{2}\right) * y \sim \mathbf{Ber}\left(\tfrac{1}{2}\right) \rangle$$

Intuitively, a probabilistic interpretation of Par is justifiable because $x :\approx \mathbf{Ber}\left(\frac{1}{2}\right)$ and $y :\approx \mathbf{Ber}\left(\frac{1}{2}\right)$ execute without interaction, so there is no correlation between their random behaviors. The lack of correlation is due in part to the lack of shared state, which makes the nondeterminism introduced by the interleaving behavior of the scheduler not observable in any way.

Bigger challenges arise when the threads *do* share state, so that the nondeterministic behavior of the scheduler *is* observable. CSL handles shared state with *resource invariants*—threads can

interact with shared state as long as the invariant I is preserved by every atomic step. More specifically, a more general Par rule allows the invariant resources to be used in both threads; the Atom rule *opens* the invariant, as long as the program is a single atomic command a; and Share allocates an invariant that is true before and after the program execution.

$$\frac{I \vdash \langle \varphi_1 \rangle\ C_1\ \langle \psi_1 \rangle \qquad I \vdash \langle \varphi_2 \rangle\ C_2\ \langle \psi_2 \rangle}{I \vdash \langle \varphi_1 * \varphi_2 \rangle\ C_1 \parallel C_2\ \langle \psi_1 * \psi_2 \rangle}\textsc{Par} \qquad \frac{\vdash \langle \varphi * \lceil I \rceil \rangle\ a\ \langle \psi * \lceil I \rceil \rangle}{I \vdash \langle \varphi \rangle\ a\ \langle \psi \rangle}\textsc{Atom}$$

$$\frac{I \vdash \langle \varphi \rangle\ C\ \langle \psi \rangle}{\vdash \langle \varphi * \lceil I \rceil \rangle\ C\ \langle \psi * \lceil I \rceil \rangle}\textsc{Share}$$

Under the probabilistic interpretation of $*$ from Chapter 3, the Par rule is valid when the shared state described by the invariant I is *deterministic*, for example, in the following program, where z has a fixed value, which is read from both threads.

$$z \mapsto 1 \vdash \langle \lceil x \mapsto - \rceil * \lceil y \mapsto - \rceil \rangle\ x :\approx \mathbf{Ber}\left(\tfrac{z}{2}\right) \parallel y :\approx \mathbf{Ber}\left(\tfrac{z}{2}\right)\ \langle (x \sim \mathbf{Ber}\left(\tfrac{1}{2}\right)) * (y \sim \mathbf{Ber}\left(\tfrac{1}{2}\right)) \rangle$$

However, with nondeterministic or randomized shared state, reasoning about programs becomes more complicated because correlations can be introduced in subtle ways. We will see a few such representative scenarios, and how they are handled in pcOL.

7.1.2 Handling Randomized Shared State with Conditioning

When shared state is randomized, compositional reasoning about outcomes becomes essential. As an example, consider the following program, where x and y read from the shared randomized variable z in different threads.

$$z :\approx \mathbf{Ber}\left(\tfrac{1}{2}\right) \fatsemi (x := z \parallel y := 1 - z) \tag{7.1}$$

Here, x and y are clearly not independently distributed, since both are derived from z. So it seems dubious that Par could be used to reason about program (7.1). The key observation is that x and y are *conditionally* independent on z, so we can compositionally reason about the threads by first breaking down the outcomes of the sampling operation such that z is deterministic in each case, similarly to how we have handled randomized control flow in prior chapters.

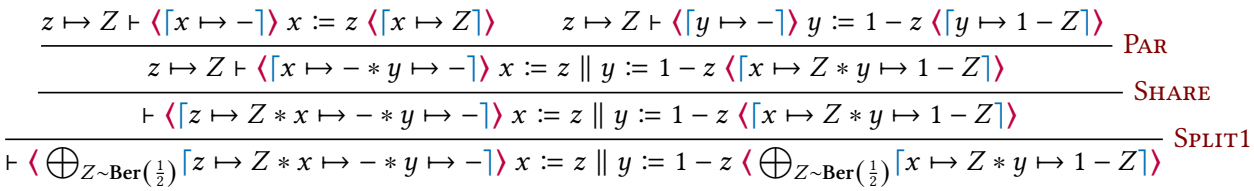


Figure 7.1: Sketch of the derivation for program (7.1).

By considering each possible outcome of the sampling operation, we end up in a situation where z is deterministic, and therefore the Par rule can apply. After executing the sampling operation, we get the assertion $(z \sim \mathbf{Ber}(\frac{1}{2})) * \lceil x \mapsto - * y \mapsto - \rceil$, which is equivalent via the distributive law to $\bigoplus_{Z \sim \mathbf{Ber}(\frac{1}{2})} \lceil z \mapsto Z * x \mapsto - * y \mapsto - \rceil$. From there, the derivation for the remainder of the program is sketched in Figure 7.1.

Before applying the Par rule, we need to use the Split1 rule from pOL and dOL to do case analysis on the value of Z. In the premise of Split1, we have that $\lceil z \mapsto Z \rceil$, so z is now deterministic and Z is unbound (and thus implicitly universally quantified). Since z is accessed in both threads, the next step is to use Share to move the information about z into the invariant. Finally, we simply use Par to conclude that $\lceil x \mapsto Z * y \mapsto 1 - Z \rceil$ for each outcome of the initial coin flip.

Ultimately, the conclusion of Split1 has the postcondition $\bigoplus_{Z \sim \mathbf{Ber}(\frac{1}{2})} \lceil x \mapsto Z * y \mapsto 1 - Z \rceil$. In other words, x and y are independent inside the scope of the $\bigoplus_{Z \sim \mathbf{Ber}(\frac{1}{2})}$, where they are deterministic. So, although x and y are clearly correlated, applications of Par and Split1 in a careful order allow us to obtain an expressive specification for the program, which still recovers the full joint distribution over x and y. The pattern of conditioning on randomized shared state will arise frequently throughout the remaining examples.

7.1.3 Taming Nondeterminism with Precise Assertions

In the previous section, we saw a strategy for compositional concurrency reasoning, but it was limited to scenarios where the shared state could be made deterministic via case analysis. In other words, the nondeterministic scheduling order could not affect the outcome of the program.

The effects of scheduling become observable when threads *mutate* shared state, because different interleavings can cause the shared variables to take on different values throughout the program execution. Like standard CSL, we need to account for the ways in which different program interleavings can nondeterministically result in different outcomes. For example, in the following program, the value of y is effectively nondeterministic, even though it is obtained by observing x immediately after x is assigned to be 1.

$$x \coloneqq 1 \fatsemi y \coloneqq x \;\;\Big\|\;\; x \coloneqq 0$$

However, unlike standard CSL, in pcOL we also need to account for the probabilistic independence guarantees of separation. As we saw in Section 5.4.2, reasoning about nondeterminism in dOL can invalidate the strong frame rule. In a similar way, certain types of observations about nondeterministic shared state can also invalidate Par.

As an example, consider the following program. Although both threads independently sample from Bernoulli distributions, the parameter of the Bernoulli distribution in the second thread depends on shared state, which is updated in the first thread.

$$z \coloneqq 1 \fatsemi x :\approx \mathbf{Ber}\left(\tfrac{1}{2}\right) \fatsemi z \coloneqq x \;\;\Big\|\;\; y :\approx \mathbf{Ber}\left(\tfrac{z}{2}\right) \tag{7.2}$$

Using the invariant $z \in \{0,1\}$, we could derive the following specification for the first thread. Since all the operations on z clearly satisfy the invariant, the specification effectively only dictates that x is distributed according to $\mathbf{Ber}\left(\frac{1}{2}\right)$.

$$z \in \{0,1\} \vdash \langle \lceil x \mapsto - \rceil \rangle \; z \coloneqq 1 \fatsemi x :\approx \mathbf{Ber}\left(\tfrac{1}{2}\right) \fatsemi z \coloneqq x \; \langle x \sim \mathbf{Ber}\left(\tfrac{1}{2}\right) \rangle$$

Turning to the second thread, since we know that $z \in \{0,1\}$, then $\frac{z}{2} \leq \frac{1}{2}$, and therefore we could obtain the following specification, which states that y is distributed according to a Bernoulli distribution with parameter at most $\frac{1}{2}$.

$$z \in \{0,1\} \vdash \langle \lceil y \mapsto - \rceil \rangle \; y :\approx \mathbf{Ber}\left(\tfrac{z}{2}\right) \; \langle \exists P. \lceil P \leq \tfrac{1}{2} \rceil * y \sim \mathbf{Ber}\,(P) \rangle$$

However, the Par rule from Section 7.1.1 allows us to join the two triples above to obtain the overall postcondition $x \sim \mathbf{Ber}\left(\frac{1}{2}\right) * \exists P. \lceil P \leq \frac{1}{2} \rceil * y \sim \mathbf{Ber}\,(P)$, which is not sound since the

scheduler can easily cause x and y to be correlated. Indeed, by selecting the interleaving where the entire first thread is executed before the second thread, the program effectively becomes $x :\approx \mathbf{Ber}\left(\frac{1}{2}\right) \fatsemi y :\approx \mathbf{Ber}\left(\frac{x}{2}\right)$. The correlation in that interleaving is obvious by inspecting conditional probabilities: $\mathbb{P}[y \mapsto 0 \mid x \mapsto 0] = 1$, whereas $\mathbb{P}[y \mapsto 0 \mid x \mapsto 1] = \frac{1}{2}$.

To ensure that the correlations between the local states of the two threads do not invalidate the Par rule in pcOL, we additionally require the postconditions of each thread to be *precise* (Definition 3.10), essentially meaning that they exactly determine the probability of each measurable event. Clearly, the postcondition that we gave to the second thread of (7.2) was not precise, since the parameter of the Bernoulli distribution was existentially quantified. Hence, the application of Par was unsound. Instead, we could use the following triple that cannot measure the probabilistic effects of nondeterminism in any way.

$$z \in \{0,1\} \vdash \langle \lceil y \mapsto - \rceil \rangle \; y :\approx \mathbf{Ber}\left(\tfrac{z}{2}\right) \; \langle \lceil y \in \{0,1\} \rceil \rangle$$

Just as we saw in Section 3.1.3, moving to the pure assertion $\lceil y \in \{0,1\} \rceil$ allows us to separate the information about x and y without violating the independence guarantees of the separating conjunction. As such, we could derive the following sound specification for (7.2).

$$z \in \{0,1\} \vdash \langle \lceil x \mapsto - * y \mapsto - \rceil \rangle \quad \begin{array}{l} z :\coloneqq 1 \fatsemi \\ x :\approx \mathbf{Ber}\left(\frac{1}{2}\right) \fatsemi \\ z :\coloneqq x \end{array} \quad \Big\| \quad y :\approx \mathbf{Ber}\left(\tfrac{z}{2}\right) \quad \langle x \sim \mathbf{Ber}\left(\tfrac{1}{2}\right) * \lceil y \in \{0,1\} \rceil \rangle$$

Once again, the use of an an almost sure assertion makes independence of the measurable events trivial, for example:

$$\mathbb{P}[x \mapsto 1 * y \in \{0,1\}] \quad = \quad \frac{1}{2} \quad = \quad \frac{1}{2} \cdot 1 \quad = \quad \mathbb{P}[x \mapsto 1] \cdot \mathbb{P}[y \in \{0,1\}]$$

Note that the precision requirement in the Par rule is very similar to the precision requirement in the Strengthen rule of dOL; in both cases, precision ensures that the nondeterministic behavior of the scheduler is not measurable in any way.

7.2 INVARIANT SENSITIVE SEMANTICS

In the prior section, we saw how the rules of PCOL can be applied for expressive probabilistic reasoning. We now turn to formalizing the metatheory, and discuss the semantic tools required to justify the soundness of the PCOL inference rules. Soundness of the PAR rule, and soundness of PCOL more generally, essentially requires that each triple $I \vdash \langle \varphi \rangle\ C\ \langle \psi \rangle$ applies not only to C running in isolation, but also to C with other threads composed in parallel. For example, consider the simple parallel program below:

$$x := 1 \fatsemi y := x \ \Big\|\ x := 0 \tag{7.3}$$

In the left thread, y is immediately set to x after x is set to 1, so the pomsets with formula model from Chapter 6 tells us that x and y are both 1 after running the thread in isolation.

$$\mathcal{L}\left(\llbracket x := 1 \fatsemi y := x \rrbracket_{pom}\right)(\sigma) = \left\{\left[\sigma[x := 1, y := 1] \mapsto 1\right]\right\}$$

But the semantics above does not take into account the fact that the program is running in a global context where shared state could be modified by other threads. Therefore, it cannot readily be used to justify the soundness of the PAR rule. In fact, the behavior of the thread depends on the other threads that are running in parallel. In the case above, the second thread is racing to update x to be 0, therefore the command $y := x$ is effectively a nondeterministic update of y to be 0 or 1.

As we saw in Section 7.1.1, we reason about shared state via *invariants*—assertions that must be preserved by every atomic action. To model the ways in which shared state may be modified by other threads, we define an *invariant sensitive semantics*, in which the scheduler may alter shared state before executing each atomic action. In the case of program (7.3), the invariant is simply $x \in \{0, 1\}$, and the invariant-sensitive execution of the action $y := x$ is given below, where the extensional interpretation of assertions $(\!|-|\!)$ was defined in Section 2.4.1.

$$\llbracket y := x \rrbracket_{act}^{(\!|x \in \{0,1\}|\!)}(\sigma) = \left\{ \left[\begin{array}{ll} \sigma[x := 1, y := 1] & \mapsto p \\ \sigma[x := 0, y := 0] & \mapsto 1 - p \end{array} \right] \;\middle|\; p \in [0, 1] \right\}$$

The scheduler can set x to be 0 or 1 before executing the action, and so the invariant sensitive semantics includes options where x and y are both 0 or both 1. Since we are still working in the convex powerdomain model, nondeterminism is represented as a convex choice, therefore the scheduler's choice of value for x can be made probabilistically. Even without a probabilistic scheduler, the convex choice would still be needed to model invariants, since shared state can be updated probabilistically, for example, in the modification of program (7.3) below.

$$x \coloneqq 1 \fatsemi y \coloneqq x \;\; \Big\| \;\; x :\approx \mathbf{Ber}\left(\tfrac{1}{2}\right)$$

The convex choice in $[\![y \coloneqq x]\!]_{act}^{(\!|x \in \{0,1\}|\!)}(\sigma)$ still captures the behavior of $y \coloneqq x$ in the presence of the new randomized thread.

To formalize the semantics, we use *semantic invariants*, which are finite sets of memories $\mathcal{I} \subseteq_{\mathsf{fin}} \mathsf{Mem}[U]$ that represent the legal values of shared state. Invariants must be finite in order to avoid issues related to unbounded nondeterminism, as discussed in Section 5.2.1. However, finite invariants are sufficient for a wide variety of verification tasks, such as the ones in Chapter 8.

The invariant-sensitive model is based on a family of semantic functions for actions, indexed by a semantic invariant: $[\![a]\!]_{act}^{\mathcal{I}} : \mathsf{Mem}[V] \to \mathcal{C}(\mathsf{Mem}[V])$, where $\mathcal{I} \subseteq_{\mathsf{fin}} \mathsf{Mem}[U]$ and $U \subseteq V$.

$$[\![a]\!]_{act}^{\mathcal{I}} \triangleq \left(\mathsf{check}^{\mathcal{I}}\right)^{\dagger} \circ [\![a]\!]_{act}^{\dagger} \circ \left(\mathsf{replace}^{\mathcal{I}}\right)^{\dagger} \circ \mathsf{check}^{\mathcal{I}}$$

$$\mathsf{check}^{\mathcal{I}}(\sigma) \triangleq \begin{cases} \eta(\sigma) & \text{if } \pi_U(\sigma) \in \mathcal{I} \\ \bot_{\mathcal{C}} & \text{if } \pi_U(\sigma) \notin \mathcal{I} \end{cases} \qquad \mathsf{replace}^{\mathcal{I}}(\sigma) = \mathop{\&}_{\tau \in \mathcal{I}} \eta(\pi_{V \setminus U}(\sigma) \uplus \tau)$$

In the invariant sensitive semantics, $\mathcal{I}$ is first checked to ensure that the current state satisfies the invariant. Next, a new valid state (or distribution thereof) is chosen to replace the shared portion of memory, simulating a parallel thread that alters the shared state at that step. The standard action semantics is then executed, followed by another check to ensure that the invariant still holds. If the invariant is ever violated, then $\bot_{\mathcal{C}}$ is returned to indicate that the execution is faulty. Letting $\mathsf{emp} \in \mathsf{Mem}[\emptyset]$ be the empty memory, we remark that $[\![a]\!]_{act} = [\![a]\!]_{act}^{\{\mathsf{emp}\}}$, meaning that invariant sensitive execution using the empty invariant is equal to normal execution, since $\mathsf{check}^{\{\mathsf{emp}\}} = \mathsf{replace}^{\{\mathsf{emp}\}} = \eta$.

The invariant sensitive semantics is monotonic: adding more variables into the invariant can only increase the set of nondeterministic outcomes.

Lemma 7.1 (Invariant Monotonicity for Actions)**.** *For any action $a \in act$ and semantic invariants $\mathcal{I} \subseteq_{\mathsf{fin}} \mathsf{Mem}[U]$ and $\mathcal{J} \subseteq_{\mathsf{fin}} \mathsf{Mem}[V]$ such that $U \cap V = \emptyset$:*

$$[\![a]\!]^{\mathcal{I}} \subseteq^{\bullet} [\![a]\!]^{\mathcal{I} * \mathcal{J}}$$

Where $\subseteq^{\bullet}$ is the pointwise extension of $\subseteq$ as defined in Lemma 2.9.

Proof. Note that the conclusion is equivalent to $[\![a]\!]^{\mathcal{I}*\mathcal{J}} \sqsubseteq_C^{\bullet} [\![a]\!]^{\mathcal{I}}$ for all σ. If $\pi_{U\cup V}(\sigma) \notin \mathcal{I} * \mathcal{J}$, then $[\![a]\!]^{\mathcal{I}*\mathcal{J}}(\sigma) = \bot_C$, therefore we trivially have that:

$$[\![a]\!]^{\mathcal{I}*\mathcal{J}}(\sigma) = \bot_C \sqsubseteq_C [\![a]\!]^{\mathcal{I}}(\sigma)$$

Now, suppose that $\pi_{U\cup V}(\sigma) \in \mathcal{I} * \mathcal{J}$, which means that the initial invariant check passes in both sides of the inequality. Clearly, the replacement using $\mathcal{I} * \mathcal{J}$ results in more nondeterministic possibilities compared to $\mathcal{I}$, therefore the inequality holds in that case too. □

Monotonicity is apparent in the execution $y := x$ from (7.3). Suppose that σ is a memory such that $\sigma(x) = 1$, then $[\![y := x]\!]_{act}(\sigma)$ simply results in y being assigned to 1, whereas $[\![y := x]\!]_{act}^{(\!|x\in\{0,1\}|\!)}(\sigma)$ sets x and y to 0 or 1 probabilistically. Choosing $p = 1$, the single outcome of $[\![y := x]\!]_{act}(\sigma)$ is clearly contained in the set obtained by invariant sensitive execution.

$$\begin{aligned}
[\![y := x]\!]_{act}(\sigma) &= \left\{\left[\sigma[y := 1] \mapsto 1\right]\right\} \\
&\subseteq \left\{ \begin{bmatrix} \sigma[x := 1, y := 1] & \mapsto p \\ \sigma[x := 0, y := 0] & \mapsto 1 - p \end{bmatrix} \;\middle|\; p \in [0,1] \right\} = [\![y := x]\!]_{act}^{(\!|x\in\{0,1\}|\!)}(\sigma)
\end{aligned}$$

Now, we extend invariant sensitive execution from actions to any commands by also indexing the linearization function by a semantic invariant $\mathcal{L}^{\mathcal{I}}$, indicating which semantic function to use for actions. So $\mathcal{L}^{\mathcal{I}}$ is defined in exactly the same was as $\mathcal{L}$ from Section 6.4, except for the selection of which function to use for $[\![-]\!]_{act}$. Returning to program (7.3), we get the following invariant sensitive semantics for the first thread.

$$\mathcal{L}^{(\!|x\in\{0,1\}|\!)}\left([\![x := 1 \fatsemi y := x]\!]_{pom}\right)(\sigma) = \left\{ \begin{bmatrix} \sigma[x := 1, y := 1] & \mapsto p \\ \sigma[x := 0, y := 0] & \mapsto 1 - p \end{bmatrix} \;\middle|\; p \in [0,1] \right\}$$

As with the invariant sensitive action execution, the value of y is now nondeterministically selected by the scheduler. Moreover, the possible distributions over the thread's local state—*i.e.*, the variable y—no longer change when we compose the two threads of (7.3) in parallel, which underlies the ability to reason about the two threads compositionally. Invariant sensitive execution effectively reduces concurrent execution into standard nondeterminism.

The linearized state transformer is ideal for modeling a program logic, where programs are specified in terms of preconditions and postconditions. As shown by Lemma 6.10, linearization of non-parallel programming constructs is well-behaved; it is compositional with respect to sequencing, if statements, and while loops. In fact, Theorem 6.11 showed that the parallel-free fragment of the linearized model is equivalent to the model of DOL, and therefore the metatheory for standard commands carries over exactly from DOL to PCOL. As we will see formally in the next section, $\mathcal{L}^{(\!|x \in \{0,1\}|\!)}$ supports the soundness of the triple below.

$$x \in \{0,1\} \vdash \langle\lceil y \mapsto - \rceil\rangle\ x := 1 \fatsemi y := x\ \langle\lceil y \in \{0,1\}\rceil\rangle$$

We omit the indexing invariant $\mathcal{I}$ and just write $\mathcal{L}$ when $\mathcal{I} = \{\mathsf{emp}\}$. When the invariant is $\{\mathsf{emp}\}$, the scheduler does not simulate any mutations performed by other threads, since all actions are evaluated according to $[\![a]\!]_{act}^{\{\mathsf{emp}\}} = [\![a]\!]_{act}$, so $\mathcal{L}([\![C]\!]_{pom})(\sigma)$ can be viewed as the *true* semantics of the program. Similar to action evaluation, linearization is also monotonic with respect to invariants.

Lemma 7.2 (Invariant Monotonicity)**.** *For any pomset $\boldsymbol{\alpha} \in pom$ and invariants $\mathcal{I} \subseteq_{\mathsf{fin}} \mathsf{Mem}[U]$ and $\mathcal{J} \subseteq_{\mathsf{fin}} \mathsf{Mem}[V]$ such that $U \cap V = \emptyset$:*

$$\mathcal{L}(\boldsymbol{\alpha})^{\mathcal{I}} \subseteq^{\bullet} \mathcal{L}(\boldsymbol{\alpha})^{\mathcal{I} * \mathcal{J}}$$

Proof. Follows immediately from monotonicity of linearization [Zilberstein et al. 2025a, Corollary G.4] and Lemma 7.1, see Lemma 5.3 of Zilberstein et al. [2026] for more details. □

Lemma 7.2 guarantees that an invariant only adds new behaviors, so safety properties about $\mathcal{L}^{\mathcal{I}}([\![C]\!]_{pom})$ automatically carry over to $\mathcal{L}([\![C]\!]_{pom})$, justifying the soundness of SHARE.

7.3 PROBABILISTIC CONCURRENT OUTCOME LOGIC

Probabilistic Concurrent Outcome Logic (PCOL) specifications are given as triples of the form $I \vDash^{\text{PCOL}}_m \langle\varphi\rangle\ C\ \langle\psi\rangle$, where φ and ψ are the extended outcome assertions from Figure 5.3, $C \in cmd_{\text{PC}}$ is a PCCL program, I is a basic separation logic assertion (Figure 2.6), and $m \in \{\text{s}, \text{w}\}$ is a mode. The meaning of PCOL triples are exactly like their DOL counterparts, except that the program C is interpreted using an invariant sensitive linearization semantics $\mathcal{L}^{I}(\llbracket C \rrbracket_{pom})$.

Since invariant sensitive execution requires the invariant states to be drawn from a finite set, I must be a *finitary* basic assertion; formally, $\text{finitary}(I)$ iff $(\!|I|\!)_\Gamma$ is a finite set for any context Γ. The formal validity definition of PCOL triples is below, which is exactly the same as that of DOL triples except that the semantic function is now $\mathcal{L}^{(\!|I|\!)_\Gamma}(\llbracket - \rrbracket_{pom})$ instead of $\llbracket - \rrbracket_C$ and $\lceil I \rceil$ is added to the precondition and postcondition.

Definition 7.3 (PCOL Triples). *The PCOL triple* $I \vDash^{\text{PCOL}}_m \langle\varphi\rangle\ C\ \langle\psi\rangle$ *is valid iff for all* $\Gamma : \text{LVar} \to \text{Val}$, μ, *and probability spaces* $\mathcal{P}, \mathcal{P}_F$, *and* $\mathcal{P}' \in \mathcal{P} \diamond_m \mathcal{P}_F$ *such that* $\mathcal{P}' \preceq \mu$ *and* $\Gamma, \mathcal{P} \vDash \varphi * \lceil I \rceil$, *then:*

$$\forall \nu \in \mathcal{L}^{(\!|I|\!)_\Gamma}\left(\llbracket C \rrbracket_{pom}\right)^{\dagger}(\mu).\quad \exists \mathcal{Q}, \mathcal{Q}' \in \mathcal{Q} \diamond_m \mathcal{P}_F.\qquad \mathcal{Q}' \preceq \nu \quad \textit{and} \quad \Gamma, \mathcal{Q} \vDash \psi * \lceil I \rceil$$

Just as DOL is a conservative extension of POL, the definition above gives us that PCOL is also a conservative extension of DOL.

Theorem 7.4. *PCOL is a conservative extension of DOL. That is, for any PDCL program* $C \in cmd_{PD}$:

$$\text{emp} \vDash^{\text{PCOL}}_m \langle\varphi\rangle\ C\ \langle\psi\rangle \qquad \textit{iff} \qquad \vDash^{\text{DOL}}_m \langle\varphi\rangle\ C\ \langle\psi\rangle$$

Proof. Since the invariant I specialized to emp and $\varphi * \lceil \text{emp} \rceil \Leftrightarrow \varphi$ for any assertion φ, we get that $\text{emp} \vDash^{\text{PCOL}}_m \langle\varphi\rangle\ C\ \langle\psi\rangle$ is definitionally equal to:

$$\begin{array}{l}\forall \Gamma, \mu, \mathcal{P}, \mathcal{P}_F, \mathcal{P}' \in \mathcal{P} \diamond_m \mathcal{P}_F.\\ \quad \mathcal{P} \otimes \mathcal{P}_F \preceq \mu \quad \wedge \quad \Gamma, \mathcal{P} \vDash \varphi \quad \Longrightarrow \\ \qquad \forall \nu \in \mathcal{L}\left(\llbracket C \rrbracket_{pom}\right)^{\dagger}(\mu).\quad \exists \mathcal{Q}, \mathcal{Q}' \in \mathcal{Q} \diamond_m \mathcal{P}_F.\quad \mathcal{Q} \otimes \mathcal{P}_F \preceq \nu \quad \wedge \quad \Gamma, \mathcal{Q} \vDash \psi\end{array}$$

By Theorem 6.11, we know that $\mathcal{L}\left(\llbracket C \rrbracket_{pom}\right) = \llbracket C \rrbracket_C$, and by rewriting that equality in the formula above, we get exactly $\vDash^{\text{DOL}}_m \langle\varphi\rangle\ C\ \langle\psi\rangle$. □

As a corollary, PCOL is also a conservative extension of POL.

Corollary 7.5. *PCOL is a conservative extension of POL. That is, for any PCL program $C \in cmd_P$:*

$$\text{emp} \vDash_s^{\text{PCOL}} \langle \varphi \rangle\ C\ \langle \psi \rangle \qquad \textit{iff} \qquad \vDash^{\text{POL}} \langle \varphi \rangle\ C\ \langle \psi \rangle$$

Proof. Follows immediately from Theorems 7.4 and 5.8. □

In the remainder of this section, we will present inference rules for deriving PCOL triples. We will henceforth omit the PCOL superscript and simply write $I \vDash_m \langle \varphi \rangle\ C\ \langle \psi \rangle$ to mean that a PCOL triple is valid. We write $I \vdash_m \langle \varphi \rangle\ C\ \langle \psi \rangle$ to mean that a triple is derivable using the PCOL rules. As with DOL, we will also omit the mode m when $m = \text{s}$. We will also omit the invariant I when $I = \text{emp}$. Following from Theorem 7.4 and Corollary 7.5, the rules of POL and DOL carry over to PCOL, but we also add new rules for reasoning about concurrency. The soundness of all three logics is discussed in Section 7.4.

7.3.1 Carryover Rules

Following from Theorem 7.4, the reasoning principles in PCOL carry over exactly from DOL. The one caveat is that there is now an additional component to the judgements, namely the invariant I. In DOL rules that have premises, the invariant must match in all premises and the conclusion. Apart from that, the inference rules for analyzing commands carry over exactly from DOL to PCOL, as shown in Figure 7.2.

The structural rules also carry over from DOL to PCOL, as shown in Figure 7.3. In addition to requiring the invariant to match between premises and conclusion, the splitting rules additionally require that the bound variable X is not free in I. If X were free in I, then those rules would be unsound, since the X that is referenced in the invariant would become the same X as is referenced in the precondition and postcondition.

Note additionally that in the rule of Consequence, the invariant cannot be strengthened or weakened, as that would violate the contract between the local thread and other threads that

$$\frac{}{I \vdash_m \langle\varphi\rangle\ \mathbf{skip}\ \langle\varphi\rangle}\textsc{Skip} \qquad \frac{I \vdash_m \langle\varphi\rangle\ C_1\ \langle\vartheta\rangle \qquad I \vdash_m \langle\vartheta\rangle\ C_2\ \langle\psi\rangle}{I \vdash_m \langle\varphi\rangle\ C_1 \fatsemi C_2\ \langle\psi\rangle}\textsc{Seq}$$

$$\frac{\varphi \Rightarrow \lceil b \mapsto 1\rceil \qquad I \vdash_m \langle\varphi\rangle\ C_1\ \langle\psi\rangle}{I \vdash_m \langle\varphi\rangle\ \mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \langle\psi\rangle}\textsc{IfT} \qquad \frac{\varphi \Rightarrow \lceil b \mapsto 0\rceil \qquad I \vdash_m \langle\varphi\rangle\ C_2\ \langle\psi\rangle}{I \vdash_m \langle\varphi\rangle\ \mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \langle\psi\rangle}\textsc{IfF}$$

$$\frac{I \vdash_m \langle\varphi\rangle\ C_1\ \langle\psi_1\rangle \quad I \vdash_m \langle\varphi\rangle\ C_2\ \langle\psi_2\rangle \quad \varphi \Rightarrow \lceil e \mapsto E\rceil \quad \psi_1 \Rightarrow \lceil e' \mapsto 1\rceil \quad \psi_2 \Rightarrow \lceil e' \mapsto 0\rceil}{I \vdash_m \langle\varphi\rangle\ C_1 \oplus_e C_2\ \langle\psi_1 \oplus_E \psi_2\rangle}\textsc{Prob}$$

$$\frac{I \vdash_w \langle\lceil P\rceil\rangle\ C_1\ \langle\psi\rangle \quad I \vdash_w \langle\lceil P\rceil\rangle\ C_2\ \langle\psi\rangle \quad \mathrm{convex}(\psi)}{I \vdash_w \langle\lceil P\rceil\rangle\ C_1 \,\&\, C_2\ \langle\psi\rangle}\textsc{Nondet}$$

$$\frac{\varphi \Rightarrow \lceil e \mapsto E\rceil \wedge (\psi * \mathsf{own}(x))}{I \vdash_m \langle\varphi\rangle\ x := e\ \langle\lceil x \mapsto E\rceil * \psi\rangle}\textsc{Assign} \qquad \frac{\varphi \Rightarrow \lceil e \mapsto E\rceil \wedge (\psi * \mathsf{own}(x))}{I \vdash_m \langle\varphi\rangle\ x :\approx d(e)\ \langle(x \sim d(E)) * \psi\rangle}\textsc{Samp}$$

$$\frac{\lceil P\rceil \Rightarrow \lceil e \mapsto E\rceil \wedge (Q * \lceil \mathsf{own}(x)\rceil) \qquad \mathrm{finite}(E)}{I \vdash_w \langle\lceil P\rceil\rangle\ x \leftarrow e\ \langle\, \&_{X\in E}\lceil Q * x \mapsto X\rceil\rangle}\textsc{NAssign}$$

$$\frac{I \vdash_m \langle\varphi * \lceil R = N > \ell\rceil\rangle\ C\ \langle(\,\&^{N-1}_{R=\ell}\,\varphi) \oplus_{\geq p} (\,\&^{h}_{R=N}\,\varphi)\rangle \quad \begin{array}{c}\varphi \Rightarrow \lceil \ell \leq R \leq h\rceil \\ \varphi * \lceil R=\ell\rceil \Rightarrow \lceil b \mapsto 0\rceil \\ \varphi * \lceil R>\ell\rceil \Rightarrow \lceil b \mapsto 1\rceil\end{array} \quad \begin{array}{c}N \notin \mathrm{vars}(\varphi) \\ 0<p\leq 1 \\ \mathrm{precise}(\varphi[\ell/R])\end{array}}{I \vdash_m \langle\,\&^{h}_{R=\ell}\,\varphi\rangle\ \mathbf{while}\ b\ \mathbf{do}\ C\ \langle\varphi[\ell/R]\rangle}\textsc{BoundedRank}$$

Figure 7.2: Inference rules for commands that carry over to pcOL from dOL.

might be running in parallel. For a more complete description of all of the carry over rules, refer to Sections 4.3 and 5.4.

7.3.2 Concurrency Rules

The new rules for pcOL pertain specifically to reasoning about concurrency and appear in Figure 7.4. The concurrency rules are inspired by Concurrent Separation Logic (CSL) [O'Hearn 2004; Brookes 2004; Vafeiadis 2011], but the metatheory behind them takes into account the more expressive assertion model, which guarantees probabilistic independence in addition to just disjointness of state.

$$\frac{I \vdash_m \langle\varphi\rangle\ C\ \langle\psi\rangle \qquad \substack{\psi \Rightarrow \lceil e \mapsto X\rceil \\ X \notin \mathsf{vars}(I)}}{I \vdash_m \langle \bigoplus_{X\sim d(E)} \varphi\rangle\ C\ \langle \bigoplus_{X\sim d(E)} \psi\rangle}\textsc{Split1} \qquad \frac{I \vdash_m \langle\varphi\rangle\ C\ \langle\psi\rangle \qquad \substack{\mathsf{convex}(\psi) \\ X \notin \mathsf{vars}(\psi, I)}}{I \vdash_m \langle \bigoplus_{X\sim d(E)} \varphi\rangle\ C\ \langle\psi\rangle}\textsc{Split2}$$

$$\frac{I \vdash_m \langle\varphi\rangle\ C\ \langle\psi\rangle \qquad \substack{\psi \Rightarrow \lceil e \mapsto X\rceil \\ X \notin \mathsf{vars}(I)}}{I \vdash_m \langle \&_{X\in E} \varphi\rangle\ C\ \langle \&_{X\in E} \psi\rangle}\textsc{NSplit1} \qquad \frac{I \vdash_m \langle\varphi\rangle\ C\ \langle\psi\rangle \qquad \substack{\mathsf{convex}(\psi) \\ X \notin \mathsf{vars}(\psi, I)}}{I \vdash_m \langle \&_{X\in E} \varphi\rangle\ C\ \langle\psi\rangle}\textsc{NSplit2}$$

$$\frac{\varphi' \Rrightarrow \varphi \qquad I \vdash_m \langle\varphi\rangle\ C\ \langle\psi\rangle \qquad \psi \Rrightarrow \psi'}{I \vdash_m \langle\varphi'\rangle\ C\ \langle\psi'\rangle}\textsc{Consequence} \qquad \frac{I \vdash_m \langle\varphi\rangle\ C\ \langle\psi\rangle}{I \vdash_m \langle\varphi *_m \vartheta\rangle\ C\ \langle\psi *_m \vartheta\rangle}\textsc{Frame}$$

$$\frac{I \vdash \langle\varphi\rangle\ C\ \langle\psi\rangle}{I \vdash_{\mathsf{w}} \langle\varphi\rangle\ C\ \langle\psi\rangle}\textsc{Weaken} \qquad \frac{I \vdash_{\mathsf{w}} \langle\varphi\rangle\ C\ \langle\psi\rangle \qquad \mathsf{precise}(\psi)}{I \vdash \langle\varphi\rangle\ C\ \langle\psi\rangle}\textsc{Strengthen}$$

$$\frac{I \vdash_{\mathsf{w}} \langle \&_{X\in E} \lceil P\rceil\rangle\ C\ \langle\psi\rangle \qquad \lceil P\rceil \Rrightarrow e \mapsto X}{I \vdash_{\mathsf{w}} \langle \lceil \exists X \in E.\ P\rceil\rangle\ C\ \langle\psi\rangle}\textsc{Exists}$$

Figure 7.3: Structural rules that carry over to pcOL from dOL.

First is the Par rule for parallel composition. Although Par looks like the analogous rule from CSL, aside from the condition about precision, the soundness of the rule is substantially more complicated due to the probabilistic interpretation of the separating conjunction. As we saw in Section 7.1.3, when two threads interact with shared state, the scheduler can induce correlation between the distributions over their local variables. As such, the outcomes of the two threads are not probabilistically independent. However, they are *observably* independent in a restricted event space. As an example, recall program (7.2), which is shown again below.

$$z \coloneqq 1 \fatsemi x :\approx \mathbf{Ber}\left(\tfrac{1}{2}\right) \fatsemi z \coloneqq x \ \Big\|\ y :\approx \mathbf{Ber}\left(\tfrac{z}{2}\right)$$

In the schedule where the entire left thread is executed before the right thread, the program is effectively equivalent to $x :\approx \mathbf{Ber}\left(\frac{1}{2}\right) \fatsemi y :\approx \mathbf{Ber}\left(\frac{x}{2}\right)$, and thus there is clearly a correlation between x and y since y is more likely to be 1 if x is also 1, as seen in the conditional probabilities below.

$$\mathbb{P}[y \mapsto 1 \mid x \mapsto 1] = \frac{1}{2} \qquad\qquad \mathbb{P}[y \mapsto 1 \mid x \mapsto 0] = 0$$

$$\frac{I \vdash \langle \varphi_1 \rangle\ C_1\ \langle \psi_1 \rangle \qquad I \vdash \langle \varphi_2 \rangle\ C_2\ \langle \psi_2 \rangle \qquad \mathsf{precise}(\psi_1, \psi_2)}{I \vdash \langle \varphi_1 * \varphi_2 \rangle\ C_1 \parallel C_2\ \langle \psi_1 * \psi_2 \rangle}\textsc{Par}$$

$$\frac{J \vdash_m \langle \varphi * \lceil I \rceil \rangle\ a\ \langle \psi * \lceil I \rceil \rangle}{I * J \vdash_m \langle \varphi \rangle\ a\ \langle \psi \rangle}\textsc{Atom} \qquad \frac{I * J \vdash_m \langle \varphi \rangle\ C\ \langle \psi \rangle \quad \mathsf{finitary}(I)}{J \vdash_m \langle \varphi * \lceil I \rceil \rangle\ C\ \langle \psi * \lceil I \rceil \rangle}\textsc{Share}$$

Figure 7.4: Concurrent Separation Logic Rules

On the other hand, a precise postcondition such as $x \sim \mathbf{Ber}\left(\frac{1}{2}\right) * \lceil y \in \{0,1\} \rceil$ is valid, since the likelihood of y taking on a particular value is not measurable at a fine enough level of granularity to observe the correlation (recall the example from Section 3.1.3).

By requiring the postconditions of each thread to be precise, we know that the probability of each measurable event must be specified exactly, so that the nondeterministic behavior of the scheduler will not be measurable in the local distributions of either thread. More formally, since ψ_1 and ψ_2 are precise, then there are unique minimal probability spaces Q_1 and Q_2 that satisfy them. The soundness proof for Par requires us to show that for any distribution ν resulting from running $C_1 \parallel C_2$, and for any events $B_1 \in \mathcal{F}_{Q_1}$ and $B_2 \in \mathcal{F}_{Q_2}$, it must be the case that $\nu(B_1 * B_2) = \mu_{Q_1}(B_1) \cdot \mu_{Q_2}(B_2)$. We do so beginning with finite pomsets.

Lemma 7.6. *For any finite pomsets $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$, events $B_1 \subseteq \mathsf{Mem}[U_1]$ and $B_2 \subseteq \mathsf{Mem}[U_2]$, invariant $\mathcal{I} \subseteq_{\mathsf{fin}} \mathsf{Mem}[V]$, and memories $\sigma_1 \in \mathsf{Mem}[U_1]$, $\sigma_2 \in \mathsf{Mem}[U_2]$, and $\tau \in \mathcal{I}$,*

$$\begin{aligned}&\mathsf{minProb}\left(\mathcal{L}^{\mathcal{I}}_{\mathsf{fin}}(\boldsymbol{\alpha} \parallel \boldsymbol{\beta})(\sigma_1 \uplus \sigma_2 \uplus \tau), B_1 * B_2 * \mathcal{I}\right)\\ &= \mathsf{minProb}\left(\mathcal{L}^{\mathcal{I}}_{\mathsf{fin}}(\boldsymbol{\alpha})(\sigma_1 \uplus \tau), B_1 * \mathcal{I}\right) \cdot \mathsf{minProb}\left(\mathcal{L}^{\mathcal{I}}_{\mathsf{fin}}(\boldsymbol{\beta})(\sigma_2 \uplus \tau), B_2 * \mathcal{I}\right)\end{aligned}$$

Where $\mathsf{minProb}(S, A) \triangleq \inf_{\mu \in S} \mu(A)$.

Proof. By induction on the length of the remaining execution (equivalently, the number of remaining scheduling steps). If the next step schedules a command from $\boldsymbol{\alpha}$, then we condition on that transition and apply the induction hypothesis to the remainder of the execution to factor the probability into the contribution from $\boldsymbol{\alpha}$ and the remaining execution of $\boldsymbol{\beta}$ (the case where

the next step comes from $\boldsymbol{\beta}$ is symmetric, yielding the desired product form). See Lemma C.3 of Zilberstein et al. [2026] for the full derivation. □

Since the proof of Lemma 7.6 is by induction on the length of the computation, it is only applicable to finite approximations of the program. However, the property also extends to unbounded computations, as the following lemma shows.

Lemma 7.7. *For any pomsets $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$, events $B_1 \subseteq \mathsf{Mem}[U_1]$ and $B_2 \subseteq \mathsf{Mem}[U_2]$, invariant $\mathcal{I} \subseteq_{\mathsf{fin}} \mathsf{Mem}[V]$, and memories $\sigma_1 \in \mathsf{Mem}[U_1]$, $\sigma_2 \in \mathsf{Mem}[U_2]$, and $\tau \in \mathcal{I}$,*

$$\forall \nu \in \mathcal{L}^{\mathcal{I}}(\boldsymbol{\alpha} \parallel \boldsymbol{\beta})(\sigma_1 \uplus \sigma_2 \uplus \tau).$$
$$\nu(B_1 * B_2 * \mathcal{I}) \geq \mathsf{minProb}\left(\mathcal{L}^{\mathcal{I}}(\boldsymbol{\alpha})(\sigma_1 \uplus \tau), B_1 * \mathcal{I}\right) \cdot \mathsf{minProb}\left(\mathcal{L}^{\mathcal{I}}(\boldsymbol{\beta})(\sigma_2 \uplus \tau), B_2 * \mathcal{I}\right)$$

Proof. Take any $\nu \in \mathcal{L}^{\mathcal{I}}(\boldsymbol{\alpha} \parallel \boldsymbol{\beta})(\sigma_1 \uplus \sigma_2 \uplus \tau)$.

$$\begin{aligned}
&\nu(B_1 * B_2 * \mathcal{I}) \\
&\geq \mathsf{minProb}\left(\mathcal{L}^{\mathcal{I}}(\boldsymbol{\alpha} \parallel \boldsymbol{\beta})(\sigma_1 \uplus \sigma_2 \uplus \tau), B_1 * B_2 * \mathcal{I}\right) \\
&= \mathsf{minProb}\left(\sup_{\boldsymbol{\alpha}' \ll \boldsymbol{\alpha}} \sup_{\boldsymbol{\beta}' \ll \boldsymbol{\beta}} \mathcal{L}^{\mathcal{I}}_{\mathsf{fin}}(\boldsymbol{\alpha}' \parallel \boldsymbol{\beta}')(\sigma_1 \uplus \sigma_2 \uplus \tau), B_1 * B_2 * \mathcal{I}\right) \\
&= \sup_{\boldsymbol{\alpha}' \ll \boldsymbol{\alpha}} \sup_{\boldsymbol{\beta}' \ll \boldsymbol{\beta}} \mathsf{minProb}\left(\mathcal{L}^{\mathcal{I}}_{\mathsf{fin}}(\boldsymbol{\alpha}' \parallel \boldsymbol{\beta}')(\sigma_1 \uplus \sigma_2 \uplus \tau), B_1 * B_2 * \mathcal{I}\right)
\end{aligned}$$

By Lemma 7.6.

$$\begin{aligned}
&= \sup_{\boldsymbol{\alpha}' \ll \boldsymbol{\alpha}} \sup_{\boldsymbol{\beta}' \ll \boldsymbol{\beta}} \mathsf{minProb}\left(\mathcal{L}^{\mathcal{I}}_{\mathsf{fin}}(\boldsymbol{\alpha}')(\sigma_1 \uplus \tau), B_1 * \mathcal{I}\right) \cdot \mathsf{minProb}\left(\mathcal{L}^{\mathcal{I}}_{\mathsf{fin}}(\boldsymbol{\beta}')(\sigma_2 \uplus \tau), B_2 * \mathcal{I}\right) \\
&= \mathsf{minProb}\left(\mathcal{L}^{\mathcal{I}}(\boldsymbol{\alpha})(\sigma_1 \uplus \tau), B_1 * \mathcal{I}\right) \cdot \mathsf{minProb}\left(\mathcal{L}^{\mathcal{I}}(\boldsymbol{\beta})(\sigma_2 \uplus \tau), B_2 * \mathcal{I}\right)
\end{aligned}$$

□

The prior two lemmas deal with cases where the start state is an individual memory, but in reality, soundness of the Par rule requires us to already start with a distribution of memories. As a final step, we prove a lemma that directly justifies soundness of the Par rule.

Lemma 7.8. *For any pairwise disjoint $U_1, U_2, V \subseteq \mathsf{Var}$ and $\boldsymbol{\alpha}_1, \boldsymbol{\alpha}_2 \in \mathit{pom}$, let $\mathcal{P}_{\mathcal{I}}$ be the trivial probability space where $\mu_{\mathcal{P}_{\mathcal{I}}}(\mathcal{I}) = 1$. For all $k \in \{1,2\}$, $\mathcal{P}_k$, $\mathcal{Q}_k$, and $\mu \in \mathcal{D}(\mathsf{Mem}[U_1 \cup U_2 \cup V])$ such that $\mathcal{P}_1 \otimes \mathcal{P}_2 \otimes \mathcal{P}_{\mathcal{I}} \preceq \mu$, if:*

$$\forall \mu_k.\ \mathcal{P}_k \otimes \mathcal{P}_{\mathcal{I}} \preceq \mu_k \implies \forall \nu_k \in \mathcal{L}^{\mathcal{I}}(\boldsymbol{\alpha}_k)^{\dagger}(\mu_k).\ \mathcal{Q}_k \otimes \mathcal{P}_{\mathcal{I}} \preceq \nu_k \tag{7.4}$$

Then:

$$\forall \nu \in \mathcal{L}^{\mathcal{I}}(\boldsymbol{\alpha}_1 \parallel \boldsymbol{\alpha}_2)^{\dagger}(\mu). \quad \mathcal{Q}_1 \otimes \mathcal{Q}_2 \otimes \mathcal{P}_{\mathcal{I}} \preceq \nu$$

Proof. Let $(A_{1,i})_{i \in I_1}$ and $(A_{2,i})_{i \in I_2}$ be the generators of $\mathcal{P}_1$ and $\mathcal{P}_2$, respectively, as witnessed by Lemma 3.6. We start by showing that for every $k \in \{1,2\}$, $i \in I_k$, and $B \in \mathcal{F}_{\mathcal{Q}_k}$:

$$\exists p_{B,k,i} \in [0,1]. \qquad \forall \sigma \in A_{k,i}.\ \forall \tau \in \mathcal{I}.\ \forall \nu \in \mathcal{L}^{\mathcal{I}}(\boldsymbol{\alpha}_k)(\sigma \uplus \tau). \qquad \nu(B * \mathcal{I}) = p_{B,k,i} \tag{7.5}$$

First, for every $j \in I_k$, fix an arbitrary $\sigma_j \in A_{k,j}$, $\tau_j \in \mathcal{I}$, and $\nu_j \in \mathcal{L}^{\mathcal{I}}(\boldsymbol{\alpha}_k)(\sigma_j \uplus \tau_j)$ and let $p_{B,k,i} = \nu_i(B * \mathcal{I})$. Now, take any $\sigma \in A_{k,i}$ and $\tau \in \mathcal{I}$ and define two distributions as follows:

$$\mu_1(\sigma_j \uplus \tau_j) \triangleq \mu_{\mathcal{P}_k}(A_{k,j}) \qquad\qquad \mu_2(\rho) \triangleq \begin{cases} \mu_{\mathcal{P}_k}(A_{k,i}) & \text{if } \rho = \sigma \uplus \tau \\ \mu_1(\rho) & \text{if } \exists j \neq i.\ \rho = \sigma_j \uplus \tau_j \end{cases}$$

So μ_1 places all of the probability mass of the event $A_{k,j}$ at the single point $\sigma_j \uplus \tau_j$ for each $j \in I_k$, and μ_2 does the same, but the mass of $A_{k,i}$ is placed specifically at $\sigma \uplus \tau$. By construction, $\mathcal{P}_k \otimes \mathcal{P}_{\mathcal{I}} \preceq \mu_1$ and $\mathcal{P}_k \otimes \mathcal{P}_{\mathcal{I}} \preceq \mu_2$. Now, take any $\nu \in \mathcal{L}^{\mathcal{I}}(\boldsymbol{\alpha}_k)(\sigma \uplus \tau)$ and let:

$$\nu_1 \triangleq \sum_{j \in I_k} \mu_{\mathcal{P}_k}(A_{k,j}) \cdot \nu_j \qquad\qquad \nu_2 \triangleq \mu_{\mathcal{P}_k}(A_{k,i}) \cdot \nu + \sum_{j \neq i} \mu_{\mathcal{P}_k}(A_{k,j}) \cdot \nu_j$$

By construction, $\nu_1 \in \mathcal{L}^{\mathcal{I}}(\boldsymbol{\alpha}_k)^{\dagger}(\mu_1)$ and $\nu_2 \in \mathcal{L}^{\mathcal{I}}(\boldsymbol{\alpha}_k)^{\dagger}(\mu_2)$, therefore by (7.4), $\mathcal{Q}_k \otimes \mathcal{P}_{\mathcal{I}} \preceq \nu_1$ and $\mathcal{Q}_k \otimes \mathcal{P}_{\mathcal{I}} \preceq \nu_2$ and therefore $\nu_1(B * \mathcal{I}) = \nu_2(B * \mathcal{I})$, which gives us:

$$\nu_1(B * \mathcal{I}) = \nu_2(B * \mathcal{I})$$

$$\sum_{j \in I_k} \mu_{\mathcal{P}_k}(A_{k,j}) \cdot \nu_j(B * \mathcal{I}) = \mu_{\mathcal{P}_k}(A_{k,i}) \cdot \nu(B * \mathcal{I}) + \sum_{j \neq i} \mu_{\mathcal{P}_k}(A_{k,j}) \cdot \nu_j(B * \mathcal{I})$$

$$\cancel{\mu_{\mathcal{P}_k}(A_{k,i})} \cdot p_{B,k,i} + \cancel{\sum_{j \neq i} \mu_{\mathcal{P}_k}(A_{k,j}) \cdot \nu_j(B * \mathcal{I})} = \cancel{\mu_{\mathcal{P}_k}(A_{k,i})} \cdot \nu(B * \mathcal{I}) + \cancel{\sum_{j \neq i} \mu_{\mathcal{P}_k}(A_{k,j}) \cdot \nu_j(B * \mathcal{I})}$$

$$p_{B,k,i} = \nu(B * \mathcal{I})$$

We now move on to prove the main claim. Take any $\nu \in \mathcal{L}^{\mathcal{I}}(\boldsymbol{\alpha}_1 \parallel \boldsymbol{\alpha}_2)^{\dagger}(\mu)$, which must have the form $\nu = \sum_{\sigma \in \mathsf{supp}(\mu)} \mu(\sigma) \cdot \nu_\sigma$ where $\nu_\sigma \in \mathcal{L}^{\mathcal{I}}(\boldsymbol{\alpha}_1 \parallel \boldsymbol{\alpha}_2)(\sigma)$ for each σ. It just remains to show that $\mathcal{Q}_1 \otimes \mathcal{Q}_2 \otimes \mathcal{P}_{\mathcal{I}} \preceq \nu$. Take any $B_1 \in \mathcal{F}_{\mathcal{Q}_1}$ and $B_2 \in \mathcal{F}_{\mathcal{Q}_2}$:

$$\nu(B_1 * B_2 * \mathcal{I}) = \sum_{\sigma \in \mathsf{supp}(\mu)} \mu(\sigma) \cdot \nu_\sigma(B_1 * B_2 * \mathcal{I})$$

Instead of summing over $\mathsf{supp}(\mu)$, we can alternatively sum over the elements of A_1 and A_2.

$$= \sum_{i \in I_1} \sum_{j \in I_2} \sum_{\sigma_1 \in A_{1,i}} \sum_{\sigma_2 \in A_{2,j}} \sum_{\tau \in \mathcal{I}} \mu(\sigma_1 \uplus \sigma_2 \uplus \tau) \cdot \nu_{\sigma_1 \uplus \sigma_2 \uplus_\tau}(B_1 * B_2 * \mathcal{I})$$

By Lemma 7.7 and (7.5):

$$\geq \sum_{i \in I_1} \sum_{j \in I_2} \sum_{\sigma_1 \in A_{1,i}} \sum_{\sigma_2 \in A_{2,j}} \sum_{\tau \in \mathcal{I}} \mu(\sigma_1 \uplus \sigma_2 \uplus \tau) \cdot p_{B_1,1,i} \cdot p_{B_2,2,j}$$
$$= \sum_{i \in I_1} \sum_{j \in I_2} \mu(A_{1,i} * A_{2,j} * \mathcal{I}) \cdot p_{B_1,1,i} \cdot p_{B_2,2,j}$$

Since $\mathcal{P}_1 \otimes \mathcal{P}_2 \otimes \mathcal{P}_{\mathcal{I}} \preceq \mu$ and $\mu_{\mathcal{P}_{\mathcal{I}}}(\mathcal{I}) = 1$:

$$= \sum_{i \in I_1} \sum_{j \in I_2} \mu_{\mathcal{P}_1 \otimes \mathcal{P}_{\mathcal{I}}}(A_{1,i} * \mathcal{I}) \cdot \mu_{\mathcal{P}_2 \otimes \mathcal{P}_{\mathcal{I}}}(A_{2,j} * \mathcal{I}) \cdot p_{B_1,1,i} \cdot p_{B_2,2,j}$$
$$= \left(\sum_{i \in I_1} \mu_{\mathcal{P}_1 \otimes \mathcal{P}_{\mathcal{I}}}(A_{1,i} * \mathcal{I}) \cdot p_{B_1,1,i} \right) \cdot \left(\sum_{j \in I_2} \mu_{\mathcal{P}_2 \otimes \mathcal{P}_{\mathcal{I}}}(A_{2,j} * \mathcal{I}) \cdot p_{B_2,2,j} \right)$$
$$= \mu_{\mathcal{Q}_1 \otimes \mathcal{P}_{\mathcal{I}}}(B_{1,i} * \mathcal{I}) \cdot \mu_{\mathcal{Q}_2 \otimes \mathcal{P}_{\mathcal{I}}}(B_{2,j} * \mathcal{I})$$
$$= \mu_{\mathcal{Q}_1}(B_{1,i}) \cdot \mu_{\mathcal{Q}_2}(B_{2,j}) \cdot \mu_{\mathcal{P}_{\mathcal{I}}}(\mathcal{I})$$

Therefore $\nu(B_1 * B_2 * \mathcal{I}) \geq \mu_{\mathcal{Q}_1 \otimes \mathcal{Q}_2 \otimes \mathcal{P}_{\mathcal{I}}}(B_1 * B_2 * \mathcal{I})$ for all $B_1 \in \mathcal{F}_{\mathcal{Q}_1}$ and $B_2 \in \mathcal{F}_{\mathcal{Q}_2}$. However, it is impossible that any of those probabilities are strictly greater than their lower bounds, since then the cumulative probability mass in $\mathcal{Q}_1 \otimes \mathcal{Q}_2 \otimes \mathcal{P}_{\mathcal{I}}$ would be strictly greater than 1. Therefore, we have that $\nu(B_1 * B_2 * \mathcal{I}) = \mu_{\mathcal{Q}_1 \otimes \mathcal{Q}_2 \otimes \mathcal{P}_{\mathcal{I}}}(B_1 * B_2 * \mathcal{I})$, which implies that $\mathcal{Q}_1 \otimes \mathcal{Q}_2 \otimes \mathcal{P}_{\mathcal{I}} \preceq \nu$. □

The next two rules in Figure 7.4 are for interacting with invariants. The Atom rule opens the invariant by moving it into the triple as an almost sure assertion, as long as the program is a single atomic action a. Since the entire program executes atomically and I is true before and after execution, then I is true at every step of the program, therefore it is a valid invariant.

Finally, the Share rule allows a finitary almost sure assertion I to be moved into the invariant. The soundness of Share follows immediately from the invariant monotonicity property (Lemma 7.2). Since the premise of Share has a larger invariant, then the set of nondeterministic outcomes resulting from running C is larger than the set resulting from running C with the smaller invariant.

$$\mathcal{L}^{\mathcal{J}}([\![C]\!]_{pom}) \subseteq^{\bullet} \mathcal{L}^{I*\mathcal{J}}([\![C]\!]_{pom})$$

We know from the premise that the postcondition ψ holds for any distribution resulting from interpreting the program using $\mathcal{L}^{I*\mathcal{J}}([\![C]\!]_{pom})$, and thus ψ must also apply to any distribution in the smaller set resulting from $\mathcal{L}^{\mathcal{J}}([\![C]\!]_{pom})$.

7.3.3 Treating Shared State as Nondeterministic Assignment

In Section 7.2, we saw how the invariant sensitive semantic model of pcOL effectively treats the observation of shared variables as nondeterminism. We now reflect that intuition into the logic by deriving an inference rule called AssignShared for reading a value from shared memory.

$$\frac{\lceil P * I \rceil \Rightarrow \lceil e \in E \rceil \qquad \lceil P \rceil \Rightarrow \lceil \mathsf{own}(x) * Q \rceil \qquad X \notin \mathsf{vars}(Q, I)}{I \vdash_{\mathsf{w}} \langle \lceil P \rceil \rangle\ x := e\ \langle \&_{X \in E} \lceil x \mapsto X * Q \rceil \rangle}\ \textsc{AssignShared}$$

Indeed, AssignShared looks a lot like NAssign. Both rules result in a weakly frame preserving triple since the scheduler may be able to induce probabilistic correlation between the distribution over values of x and other variables in the global memory. The key difference is that in AssignShared, we can use information from the invariant I to determine what values e can range over. As before, the precondition P alone must imply that x is owned by the local thread.

Lemma 7.9. *The AssignShared rule defined above is valid.*

Proof. The derivation using rules of PCOL is shown below.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\lceil e \mapsto X \wedge \mathsf{own}(x) * Q * I\rceil \Rightarrow \lceil e \mapsto X\rceil \wedge \lceil \mathsf{own}(x) * Q * I\rceil}{\vdash_{\mathsf{w}} \langle \lceil e \mapsto X \wedge \mathsf{own}(x) * Q * I\rceil \rangle\ x := e\ \langle \lceil x \mapsto X * Q * I\rceil \rangle}\ \textsc{Assign}}{\vdash_{\mathsf{w}} \langle \&_{X\in E}\lceil e \mapsto X \wedge \mathsf{own}(x) * Q * I\rceil \rangle\ x := e\ \langle \&_{X\in E}\lceil x \mapsto X * Q * I\rceil \rangle}\ \textsc{NSplit1}}{\vdash_{\mathsf{w}} \langle \lceil \exists X \in E.\ e \mapsto X \wedge \mathsf{own}(x) * Q * I\rceil \rangle\ x := e\ \langle \&_{X\in E}\lceil x \mapsto X * Q * I\rceil \rangle}\ \textsc{Exists}}{\vdash_{\mathsf{w}} \langle \lceil P\rceil * \lceil I\rceil \rangle\ x := e\ \langle \left(\&_{X\in E}\lceil x \mapsto X * Q\rceil\right) * \lceil I\rceil \rangle}\ \textsc{Consequence}}{I \vdash_{\mathsf{w}} \langle \lceil P\rceil \rangle\ x := e\ \langle \&_{X\in E}\lceil x \mapsto X * Q\rceil \rangle}\ \textsc{Atom}$$

First, we use ATOM to bring the invariant I into the precondition and postcondition. We then apply a CONSEQUENCE to get the precondition into a form where the value of the expression e is existentially quantified. Since $\wedge$ is idempotent, we can first make two copies of $\lceil P\rceil * \lceil I\rceil$, then we apply the two consequences from the premise of ASSIGNSHARED, and finally, since $X \notin \mathsf{vars}(Q, I)$, we can combine all three of the pure assertions with an existential quantifier on the outer level.

$$\begin{aligned}\lceil P\rceil * \lceil I\rceil &\Longrightarrow \lceil P * I\rceil \wedge (\lceil P\rceil * \lceil I\rceil)\\ &\Longrightarrow \lceil e \in E\rceil \wedge (\lceil \mathsf{own}(x) * Q\rceil * \lceil I\rceil)\\ &\Longrightarrow \lceil \exists X \in E.\ e \mapsto X \wedge \mathsf{own}(x) * Q * I\rceil\end{aligned}$$

Next, we apply EXISTS to convert the pure existential quantifier into a nondeterministic outcome conjunction. The application of EXISTS forces the resulting specification to be a weakly frame preserving triple, since it makes correlations with the expression e measurable.

Now, we have the same nondeterministic outcome conjunction over the precondition and postcondition, so we apply NSPLIT1 to do case analysis on the nondeterministic value of e. Finally, we complete the derivation with a simple application of ASSIGN. □

7.4 SOUNDNESS

We now return to the question of *soundness* of the proof rules for PCOL, in addition to POL and DOL. That is, we must prove that any triple that is derivable from the rules is valid according to the relevant validity definition (4.1, 5.7 and 7.3). We begin with PCOL, and the remaining soundness

proofs will follow from the fact that PCOL is a conservative extension of POL (Corollary 7.5) and DOL (Theorem 7.4).

Theorem 7.10 (Soundness of PCOL)**.**

$$I \vdash_m \langle \varphi \rangle\ C\ \langle \psi \rangle \qquad \Longrightarrow \qquad I \vDash_m^{\text{PCOL}} \langle \varphi \rangle\ C\ \langle \psi \rangle$$

Proof. The proof is by induction on the length of the derivation. Refer to Theorem 5.2 of Zilberstein et al. [2026] for the full proof. We will focus here on the cases where the last step of the derivation is an application a concurrency rule from Figure 7.4.

- PAR.

$$\frac{I \vdash \langle \varphi_1 \rangle\ C_1\ \langle \psi_1 \rangle \qquad I \vdash \langle \varphi_2 \rangle\ C_2\ \langle \psi_2 \rangle \qquad \mathsf{precise}(\psi_1, \psi_2)}{I \vdash \langle \varphi_1 * \varphi_2 \rangle\ C_1 \parallel C_2\ \langle \psi_1 * \psi_2 \rangle}\text{PAR}$$

  Since we are using a strong triple, suppose that $\mathcal{P}_1 \otimes \mathcal{P}_2 \otimes \mathcal{P}_I \otimes \mathcal{P}_F \preceq \mu$ such that $\Gamma, \mathcal{P}_1 \vDash \varphi_1$ and $\Gamma, \mathcal{P}_2 \vDash \varphi_2$ and $\Gamma, \mathcal{P}_I \vDash \lceil I \rceil$. Without loss of generality, suppose that $\mathcal{P}_I$ is minimal. To complete the proof, we need to establish the premise of Lemma 7.8. Since ψ_1 and ψ_2 are precise, let Q_1 and Q_2 be the unique smallest probability spaces that satisfy them under Γ (note that if ψ_1 or ψ_2 is unsatisfiable, then the premise of the rule is false, so the claim holds vacuously). Take any μ_1 such that $\mathcal{P}_1 \otimes \mathcal{P}_I \preceq \mu_1$ and any $\nu_1 \in \mathcal{L}^{(\!|I|\!)_\Gamma}(\llbracket C_1 \rrbracket)^\dagger(\mu_1)$. By the premise of the PAR rule, we know that there is a $Q_1' \preceq \nu_1$ such that $\Gamma, Q_1' \vDash \psi_1 * \lceil I \rceil$, therefore $Q_1 * \mathcal{P}_I \preceq Q_1'$, and therefore we have shown that:

$$\forall \mu_1.\ \mathcal{P}_1 \otimes \mathcal{P}_I \preceq \mu_1 \implies \forall \nu_1 \in \mathcal{L}^{(\!|I|\!)_\Gamma}(\llbracket C_1 \rrbracket)^\dagger(\mu_1).\ Q_1 \otimes \mathcal{P}_I \preceq \nu_1$$

  We now perform a nearly identical argument for C_2, but we also handle the frame. Take any μ_2 such that $\mathcal{P}_2 \otimes \mathcal{P}_I \otimes \mathcal{P}_F \preceq \mu_2$ and any $\nu_2 \in \mathcal{L}^{(\!|I|\!)_\Gamma}(\llbracket C_2 \rrbracket)^\dagger(\mu_2)$. By the second premise of the PAR rule, we get that there is a Q_2' such that $Q_2' \otimes \mathcal{P}_F \preceq \nu_2$ and $\Gamma, Q_2' \vDash \psi_2 * \lceil I \rceil$. Due to precision, we know that $Q_2 * \mathcal{P}_I \preceq Q_2'$ and so $Q_2 \otimes \mathcal{P}_I \otimes \mathcal{P}_F \preceq Q_2' \otimes \mathcal{P}_F \preceq \nu_2$, so we have shown that:

$$\forall \mu_2.\ \mathcal{P}_2 \otimes \mathcal{P}_I \otimes \mathcal{P}_F \preceq \mu_2 \implies \forall \nu_2.\ \mathcal{L}^{(\!|I|\!)_\Gamma}(\llbracket C_2 \rrbracket)^\dagger(\mu_2).\ Q_2 \otimes \mathcal{P}_I \otimes \mathcal{P}_F \preceq \nu_2$$

Now, by Lemma 7.8, $\mathcal{Q}_1 \otimes \mathcal{Q}_2 \otimes \mathcal{P}_I \otimes \mathcal{P}_F \preceq \nu$ for all $\nu \in \mathcal{L}^{(\!|I|\!)_\Gamma}(\llbracket C_1 \rrbracket \parallel \llbracket C_2 \rrbracket)^\dagger(\mu)$. Let $\mathcal{Q} = \mathcal{Q}_1 \otimes \mathcal{Q}_2 \otimes \mathcal{P}_I$. So $\mathcal{Q} \otimes \mathcal{P}_F \preceq \nu$ and since $\Gamma, \mathcal{Q}_1 \vDash \psi_1$ and $\Gamma, \mathcal{Q}_2 \vDash \psi_2$, we get that $\Gamma, \mathcal{Q} \vDash \psi_1 * \psi_2 * \lceil I \rceil$, so we are done.

- Share.

$$\frac{I * J \vdash_m \langle \varphi \rangle\ C\ \langle \psi \rangle \quad \mathsf{finitary}(I)}{J \vdash_m \langle \varphi * \lceil I \rceil \rangle\ C\ \langle \psi * \lceil I \rceil \rangle}\textsc{Share}$$

Suppose that $\mathcal{P}' \in \mathcal{P} \diamond_m \mathcal{P}_F$, $\mathcal{P}' \preceq \mu$, and $\Gamma, \mathcal{P} \vDash \varphi * \lceil I \rceil * \lceil J \rceil$. Now, take any $\nu \in \mathcal{L}^{(\!|J|\!)_\Gamma}(\llbracket C \rrbracket)^\dagger(\mu)$. By Lemma 7.2, we know that:

$$\mathcal{L}^{(\!|J|\!)_\Gamma}(\llbracket C \rrbracket)^\dagger(\mu) \quad \subseteq \quad \mathcal{L}^{(\!|I|\!)_\Gamma * (\!|J|\!)_\Gamma}(\llbracket C \rrbracket)^\dagger(\mu) \quad = \quad \mathcal{L}^{(\!|I*J|\!)_\Gamma}(\llbracket C \rrbracket)^\dagger(\mu)$$

Therefore $\nu \in \mathcal{L}^{(\!|I*J|\!)_\Gamma}(\llbracket C \rrbracket)^\dagger(\mu)$ and by the induction hypothesis, we know that there exist $\mathcal{Q}$ and $\mathcal{Q}' \in \mathcal{Q} \diamond_m \mathcal{P}_F$ such that $\mathcal{Q}' \preceq \nu$ and $\Gamma, \mathcal{Q} \vDash \psi * \lceil I * J \rceil$. Since $\lceil I * J \rceil \Leftrightarrow \lceil I \rceil * \lceil J \rceil$, we get that $\Gamma, \mathcal{Q} \vDash (\psi * \lceil I \rceil) * \lceil J \rceil$, completing the proof.

- Atom.

$$\frac{J \vdash_m \langle \varphi * \lceil I \rceil \rangle\ a\ \langle \psi * \lceil I \rceil \rangle}{I * J \vdash_m \langle \varphi \rangle\ a\ \langle \psi \rangle}\textsc{Atom}$$

Suppose that $\mathcal{P}' \in \mathcal{P} \diamond_m \mathcal{P}_F$, $\mathcal{P}' \preceq \mu$, and $\Gamma, \mathcal{P} \vDash \varphi * \lceil I * J \rceil$. This means that there exist $\mathcal{P}_1$ and $\mathcal{P}_2$ such that $\mathcal{P}_1 \otimes \mathcal{P}_2 \preceq \mathcal{P}$ and $\Gamma, \mathcal{P}_1 \vDash \varphi$ and $\Gamma, \mathcal{P}_2 \vDash \lceil I * J \rceil$. Let $\mathcal{P}_I$ and $\mathcal{P}_J$ be the trivial probability spaces that satisfy $\lceil I \rceil$ and $\lceil J \rceil$, respectively, so clearly $\mathcal{P}_I \otimes \mathcal{P}_J \preceq \mathcal{P}_2$. Now, take any $\nu \in \mathcal{L}^{(\!|I*J|\!)_\Gamma}(\llbracket a \rrbracket)^\dagger(\mu)$ and note that:

$$\begin{aligned}
\mathcal{L}^{(\!|I*J|\!)_\Gamma}(\llbracket a \rrbracket)^\dagger(\mu) &= (\llbracket a \rrbracket_{act}^{(\!|I*J|\!)_\Gamma})^\dagger(\mu) \\
&= \left(\left(\mathsf{check}^{(\!|I*J|\!)_\Gamma}\right)^\dagger \circ \llbracket a \rrbracket_{act}^\dagger \circ \left(\mathsf{replace}^{(\!|I*J|\!)_\Gamma}\right)^\dagger \circ \left(\mathsf{check}^{(\!|I*J|\!)_\Gamma}\right)\right)^\dagger (\mu) \\
&= \left(\mathsf{check}^{(\!|I|\!)_\Gamma * (\!|J|\!)_\Gamma}\right)^\dagger \left(\llbracket a \rrbracket_{act}^\dagger \left(\left(\mathsf{replace}^{(\!|I|\!)_\Gamma * (\!|J|\!)_\Gamma}\right)^\dagger \left(\left(\mathsf{check}^{(\!|I|\!)_\Gamma * (\!|J|\!)_\Gamma}\right)^\dagger (\mu)\right)\right)\right) \\
&= \left(\mathsf{check}^{(\!|I|\!)_\Gamma}\right)^\dagger \left(\mathcal{L}^{(\!|J|\!)_\Gamma}(\llbracket a \rrbracket)^\dagger \left(\left(\mathsf{replace}^{(\!|I|\!)_\Gamma}\right)^\dagger \left(\left(\mathsf{check}^{(\!|I|\!)_\Gamma}\right)^\dagger (\mu)\right)\right)\right)
\end{aligned}$$

Since we already assumed that μ satisfies $I * J$, the first check does nothing.

$$= \left(\mathsf{check}^{(\!|I|\!)_\Gamma}\right)^\dagger \left(\mathcal{L}^{(\!|J|\!)_\Gamma}(\llbracket a \rrbracket)^\dagger \left(\left(\mathsf{replace}^{(\!|I|\!)_\Gamma}\right)^\dagger (\mu)\right)\right)$$

Therefore, we know that $v = (\mathsf{check}^{(\!|I|\!)_\Gamma})^\dagger(v')$ for some $v' \in \mathcal{L}^{(\!|J|\!)_\Gamma}([\![a]\!])^\dagger(\mu')$ and $\mu' \in (\mathsf{replace}^{(\!|I|\!)_\Gamma})^\dagger(\mu))$. We know that $\mathcal{P}_1 \otimes \mathcal{P}_I \otimes \mathcal{P}_J \preceq \mathcal{P}_1 \otimes \mathcal{P}_2 \preceq \mathcal{P}$, so there must be a $\mathcal{P}'' \in (\mathcal{P}_1 \otimes \mathcal{P}_I \otimes \mathcal{P}_J) \diamond_m \mathcal{P}_F$ such that $\mathcal{P}'' \preceq \mathcal{P}'$ and therefore $\mathcal{P}'' \preceq \mu$. Since $\mathcal{P}''$ only contains trivial information about I, and μ and μ' differ only in the states that satisfy I, then $\mathcal{P}'' \preceq \mu'$ too. Therefore, by the induction hypothesis, there must be $\mathcal{Q}$ and $\mathcal{Q}' \in \mathcal{Q} \diamond_m \mathcal{P}_F$ such that $\mathcal{Q}' \preceq v'$ and $\Gamma, \mathcal{Q} \vDash \psi * \lceil I \rceil * \lceil J \rceil$. This implies that $\Gamma, v' \vDash \lceil I \rceil$, and therefore $v = v'$, since the check operation does nothing. Therefore, we get that $\mathcal{Q}' \preceq v' = v$, and so we are done. □

Now that we have established the soundness of PCOL, we can move on to POL and DOL. Note that all the rules of POL and DOL appear in Figures 7.2 and 7.3 (albeit, in a more generalized form), therefore any POL (resp. DOL) derivation using the rules from Chapter 4 (resp. Chapter 5) can be replicated using the PCOL proof rules from Section 7.3. We can therefore use Theorem 7.10 to directly justify the soundness of POL and DOL. We begin with POL. By restricting the invariant I to be emp, the mode m to be s, and the program C to be a PCL program, we are able to derive triples of the form $\vdash \langle \varphi \rangle\ C\ \langle \psi \rangle$. As the following corollary shows, any such triple is valid in POL.

Corollary 7.11 (Soundness of POL)**.** *For any PCL program $C \in cmd_P$:*

$$\vdash \langle \varphi \rangle\ C\ \langle \psi \rangle \quad \Longrightarrow \quad \vDash^{\mathsf{POL}} \langle \varphi \rangle\ C\ \langle \psi \rangle$$

Proof.
$$\vdash \langle \varphi \rangle\ C\ \langle \psi \rangle \implies \vDash^{\mathsf{PCOL}} \langle \varphi \rangle\ C\ \langle \psi \rangle \quad \text{By Theorem 7.10.}$$
$$\iff \vDash^{\mathsf{POL}} \langle \varphi \rangle\ C\ \langle \psi \rangle \quad \text{By Corollary 7.5.}$$

□

If we instead restrict I to be emp and the program C to be a PDCL program, we can derive triples of the form $\vdash_m \langle \varphi \rangle\ C\ \langle \psi \rangle$, and any such triple is valid in DOL.

Corollary 7.12 (Soundness of DOL)**.** *For any PDCL program $C \in cmd_{PD}$:*

$$\vdash_m \langle \varphi \rangle\ C\ \langle \psi \rangle \quad \Longrightarrow \quad \vDash_m^{\mathsf{DOL}} \langle \varphi \rangle\ C\ \langle \psi \rangle$$

Proof.
$$\vdash_m \langle \varphi \rangle\ C\ \langle \psi \rangle \implies \vDash_m^{\mathsf{PCOL}} \langle \varphi \rangle\ C\ \langle \psi \rangle \quad \text{By Theorem 7.10.}$$
$$\iff \vDash_m^{\mathsf{DOL}} \langle \varphi \rangle\ C\ \langle \psi \rangle \quad \text{By Theorem 7.4.}$$

□

7.5 EXAMPLES

We now turn back to the running examples one final time. Compared to the examples in Section 5.5, we now modify the two programs so that nondeterminism comes from observing the values of shared variables rather than an explicit demonic choice. As such, the derivations are very similar to their prior versions, except that they now include a resource invariant and the applications of NAssign are replaced with applications of the AssignShared rule that we just derived in the previous section.

7.5.1 The Concurrent Random Walk

In Section 4.4.1 we saw a random walk program where an agent moves towards termination with probability $\frac{1}{2}$, and in Section 5.5.1 we saw a demonic variant of the random walk where an adversary could reset the agent's position in the case that it does not take a step towards termination. Now, we present a new version of the random walk where the reset position is stored in shared state rather than being chosen demonically. In effect, the reset is still nondeterministic because we do not know what other threads are running in parallel. The program representing the new concurrent random walk is shown below.

$$\begin{array}{l}\textbf{while } x > 0 \textbf{ do} \\ \quad x \coloneqq x - 1 \oplus_{\frac{1}{2}} x \coloneqq \max(x, z)\end{array}$$

The reset position is stored in the variable z. When a reset happens, the agent's position x is updated to $\max(x, z)$, which ensures that the agent will not move towards termination.

As before, the goal is to prove that the program almost surely terminates with $x \mapsto 0$, and hence the desired postcondition is $\lceil x \mapsto 0 \rceil$. The derivation is shown in Figure 7.5. Compared to the demonic version (Figure 5.9), there are essentially two changes. First, instead of $z \mapsto Z$ being a constant assertion in the precondition, z is now a shared variable governed by the resource invariant $z \in \{0, .., Z\}$. As such, the loop invariant φ no longer references z.

$$\varphi \triangleq \lceil x \mapsto R * 0 \leq R \leq \max(X, Z) \rceil$$

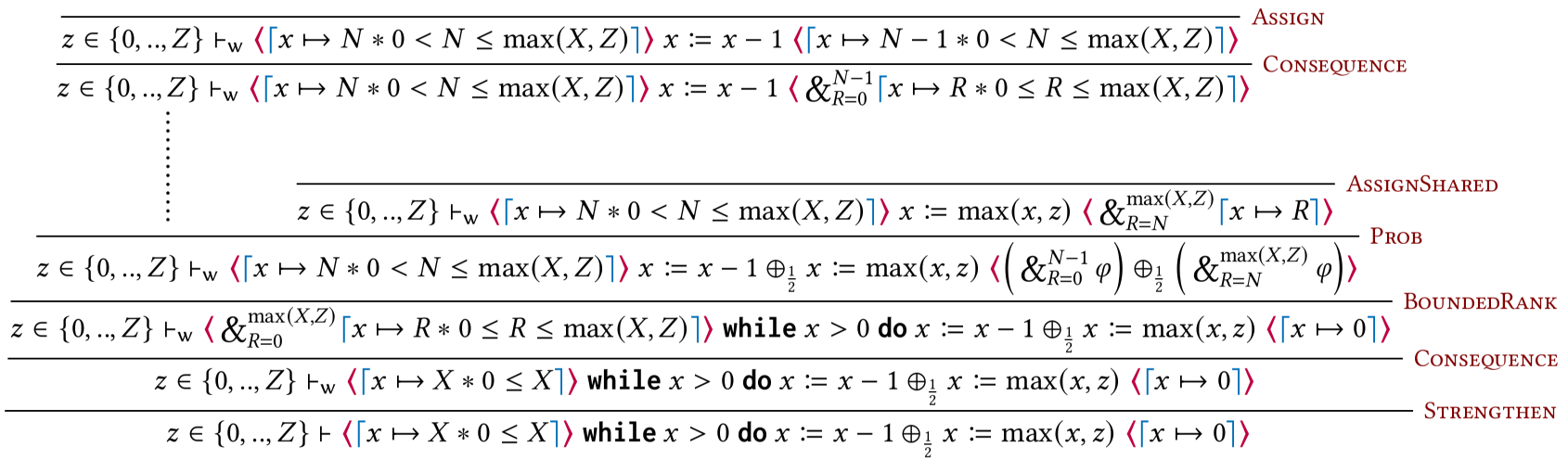


Figure 7.5: Almost sure termination proof for the concurrent random walk

The second change is that we use the derived AssignShared rule instead of NAssign to analyze the second branch of the probabilistic choice. The premise of that rule requires that:

$$\lceil x \mapsto N * 0 < N \leq \max(X, Z) * z \in \{0, .., Z\} \rceil \Rightarrow \lceil \max(x, z) \in \{N, .. \max(X, Z)\} \rceil$$

$$\lceil x \mapsto N * 0 < N \leq \max(X, Z) \rceil \Rightarrow \lceil \mathsf{own}(x) \rceil \qquad R \notin \mathsf{vars}(z \in \{0, .., Z\})$$

The second two premises clearly hold. We can prove the first implication by upper and lower bounding $\max(x, z)$. For the lower bound, we we know that $x = N$ and $x \leq \max(x, z)$, therefore $N \leq \max(x, z)$. For the upper bound, we know that $z \leq Z \leq \max(X, Z)$ and $x = N \leq \max(X, Z)$, therefore since both x and z are upper bounded by $\max(X, Z)$, then $\max(x, z) \leq \max(X, Z)$.

7.5.2 The Concurrent von Neumann Trick

As we saw in Section 4.4.2, the von Neumann [1951] trick is a randomness preserving protocol for simulating a fair coin using a biased coin. In Section 5.5.2, we worked with a demonic version of the protocol, where the bias of the coin is updated by an adversary on each round. We now present a concurrent version, where the bias is instead stored in the shared variable q and each round it is copied into the local variable p.

$$\begin{array}{l} x \coloneqq 0 \fatsemi \\ y \coloneqq 0 \fatsemi \\ \textbf{while } x = y \textbf{ do} \\ \quad p \coloneqq q \fatsemi \\ \quad x :\approx \textbf{Ber}\,(p) \fatsemi \\ \quad y :\approx \textbf{Ber}\,(p) \end{array}$$

Since parallel threads could alter q between rounds, the bias on each round is effectively chosen nondeterministically. Nevertheless, the program almost surely terminates with x being distributed like a fair coin. The derivation is shown in Figure 7.6.

Just as in the previous section, there are two differences when compared to the correctness proof for the demonic version of the von Neumann trick (Figure 5.10). First, we have added a shared variable q, and so we also must add a resource invariant $q \in [\varepsilon, 1 - \varepsilon]_\Delta$ governing the allowed values of q. As before, keeping q within the bounds ε and $1 - \varepsilon$ ensures that the

$$\frac{}{q \in [\varepsilon, 1-\varepsilon]_\Delta \vdash_{\mathsf{w}} \langle x \sim \mathbf{Ber}\,(Z) * \lceil \mathsf{own}(y) * p \mapsto Z \rceil \rangle\ y :\approx \mathbf{Ber}\,(p)\ \langle x \sim \mathbf{Ber}\,(Z) * y \sim \mathbf{Ber}\,(Z) * \lceil p \mapsto Z \rceil \rangle}\ \textsc{Samp} \tag{7.6}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{q \in [\varepsilon, 1-\varepsilon]_\Delta \vdash_{\mathsf{w}} \langle \lceil \mathsf{own}(x,y) * p \mapsto Z \rceil \rangle\ x :\approx \mathbf{Ber}\,(p)\ \langle x \sim \mathbf{Ber}\,(Z) * \lceil \mathsf{own}(y) * p \mapsto Z \rceil \rangle}\ \textsc{Samp} \quad (7.6)}{q \in [\varepsilon, 1-\varepsilon]_\Delta \vdash_{\mathsf{w}} \langle \lceil \mathsf{own}(x,y) * p \mapsto Z \rceil \rangle\ x :\approx \mathbf{Ber}\,(p) \fatsemi y :\approx \mathbf{Ber}\,(p)\ \langle x \sim \mathbf{Ber}\,(Z) * y \sim \mathbf{Ber}\,(Z) * \lceil p \mapsto Z \rceil \rangle}\ \textsc{Seq}}{q \in [\varepsilon, 1-\varepsilon]_\Delta \vdash_{\mathsf{w}} \langle \lceil (x = y) \mapsto 1 * p \mapsto Z \rceil \rangle\ x :\approx \mathbf{Ber}\,(p) \fatsemi y :\approx \mathbf{Ber}\,(p)\ \langle \varphi_0 \oplus_{\geq 2\varepsilon(1-\varepsilon)} \varphi_1 \rangle}\ \textsc{Consequence}}{q \in [\varepsilon, 1-\varepsilon]_\Delta \vdash_{\mathsf{w}} \langle \&_{Z \in [\varepsilon, 1-\varepsilon]_\Delta} \lceil (x = y) \mapsto 1 * p \mapsto Z \rceil \rangle\ x :\approx \mathbf{Ber}\,(p) \fatsemi y :\approx \mathbf{Ber}\,(p)\ \langle \varphi_0 \oplus_{\geq 2\varepsilon(1-\varepsilon)} \varphi_1 \rangle}\ \textsc{NSplit2} \tag{7.7}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{q \in [\varepsilon, 1-\varepsilon]_\Delta \vdash_{\mathsf{w}} \langle \lceil (x = y) \mapsto 1 * \mathsf{own}(p) \rceil \rangle\ p := q\ \langle \&_{Z \in [\varepsilon, 1-\varepsilon]_\Delta} \lceil (x = y) \mapsto 1 * p \mapsto Z \rceil \rangle}\ \textsc{AssignShared} \quad (7.7)}{q \in [\varepsilon, 1-\varepsilon]_\Delta \vdash_{\mathsf{w}} \langle \lceil (x = y) \mapsto 1 * \mathsf{own}(p) \rceil \rangle\ p := q \fatsemi x :\approx \mathbf{Ber}\,(p) \fatsemi y :\approx \mathbf{Ber}\,(p)\ \langle \varphi_0 \oplus_{\geq 2\varepsilon(1-\varepsilon)} \varphi_1 \rangle}\ \textsc{Seq}}{q \in [\varepsilon, 1-\varepsilon]_\Delta \vdash_{\mathsf{w}} \langle \&^1_{R=0} \varphi \rangle\ \mathbf{while}\ x = y\ \mathbf{do}\ p := q \fatsemi x :\approx \mathbf{Ber}\,(p) \fatsemi y :\approx \mathbf{Ber}\,(p)\ \langle \bigoplus_{X \sim \mathbf{Ber}\left(\frac{1}{2}\right)} \lceil x \mapsto X * y \mapsto 1 - X \rceil \rangle}\ \textsc{BoundedRank}}{q \in [\varepsilon, 1-\varepsilon]_\Delta \vdash_{\mathsf{w}} \langle \lceil x \mapsto 0 * y \mapsto 0 * \mathsf{own}(p) \rceil \rangle\ \mathbf{while}\ x = y\ \mathbf{do}\ p := q \fatsemi x :\approx \mathbf{Ber}\,(p) \fatsemi y :\approx \mathbf{Ber}\,(p)\ \langle x \sim \mathbf{Ber}\left(\frac{1}{2}\right) \rangle}\ \textsc{Consequence}}{q \in [\varepsilon, 1-\varepsilon]_\Delta \vdash \langle \lceil x \mapsto 0 * y \mapsto 0 * \mathsf{own}(p) \rceil \rangle\ \mathbf{while}\ x = y\ \mathbf{do}\ p := q \fatsemi x :\approx \mathbf{Ber}\,(p) \fatsemi y :\approx \mathbf{Ber}\,(p)\ \langle x \sim \mathbf{Ber}\left(\frac{1}{2}\right) \rangle}\ \textsc{Strengthen}$$

Figure 7.6: Correctness proof for the concurrent von Neumann trick.

adversary cannot force nontermination, and the step size Δ ensures that the interval is finite. The loop invariant φ remains exactly the same:

$$\varphi_0 \triangleq \bigoplus_{X \sim \mathbf{Ber}(\frac{1}{2})} \lceil x \mapsto X * y \mapsto \neg X \rceil \qquad \varphi_1 \triangleq \lceil (x = y) \mapsto 1 * \mathsf{own}(p) \rceil$$

$$\varphi \triangleq (\varphi_0 * \lceil R = 0 \rceil) \vee (\varphi_1 * \lceil R = 1 \rceil)$$

The second change is that we use AssignShared to analyze the read from shared state, instead of NAssign, which was used to analyze the prior nondeterministic choice. The premises of AssignShared all clearly hold:

$$\lceil (x = y) \mapsto 1 * \mathsf{own}(p) * q \in [\varepsilon, 1 - \varepsilon]_\Delta \rceil \Rrightarrow \lceil q \in [\varepsilon, 1 - \varepsilon]_\Delta \rceil$$

$$\lceil (x = y) \mapsto 1 * \mathsf{own}(p) \rceil \Rrightarrow \lceil (x = y) \mapsto 1 * \mathsf{own}(p) \rceil \qquad R \notin \mathsf{vars}((x = y) \mapsto 1, q \in [\varepsilon, 1 - \varepsilon]_\Delta)$$

The final specification is quite interesting. Although an adversary can influence the amount of time that it takes for the program to terminate, it cannot influence the probability of eventual termination. In addition, pcOL tells us not only that the final specification from Figure 7.6 holds, but also that x is probabilistically independent from the local variables of any other concurrent threads. Not only can an adversary not affect the termination probability or distribution over values of x, but it also cannot induce any correlation with x.

7.6 RELATED WORK

Logics for Probabilistic Concurrency. Polaris is a relational separation logic built on Iris [Jung et al. 2018] for reasoning about concurrent probabilistic programs [Tassarotti 2018; Tassarotti and Harper 2019]. Compared to Polaris, pcOL has two main advantages: it supports unbounded looping and it supports direct probabilistic reasoning about the distribution of outcomes. Polaris cannot handle the von Neumann trick, which involves unbounded looping. Because it is a relational logic, Polaris works by relating a randomized concurrent program to a specification program, which is randomized and nondeterministic (but not concurrent). That requires the random choices between the two programs to be coupled in lockstep.

In the concurrent shuffling example that was introduced in Section 1.2, we prove with pcOL that a sequence of many random choices results in a completely different uniform distribution of outcomes. It would be impossible to recreate this proof in Polaris, since the natural specification program for a randomized shuffle would only make a single random choice (a uniform sample over permutations), whereas the actual program makes a large sequence of coin flips. By instead picking a specification program in Polaris that also makes a similar sequence of random choices, one might be able to do the relational proof, but then would be left with the challenge of proving that such a specification program actually generated a uniform shuffle, which Polaris cannot do.

In pcOL, we combine the two steps, avoiding the need for a specification program by building quantitative reasoning tools into the logic itself. However, the approaches do have similarities. The invariant sensitive semantics implicitly captures the behavior of the specification program by converting parallel manipulation of shared state into nondeterminism, which is slightly easier since the user must only write down an invariant rather than an entire specification program. Then, rather than externally analyzing the specification program, we directly prove a quantitative postcondition. While pcOL does not support all the capabilities of Polaris (*e.g.*, ghost state, higher order state, etc.), proofs are carried out in fewer steps using a single, self-contained logic. In the future, it may be fruitful to combine the two approaches. Indeed, Bao et al. [2025] showed the advantages of supporting both relational and unary reasoning in a single probabilistic logic.

Fesefeldt et al. [2022] pursued an alternative technique for reasoning about probabilistic concurrent programs, based on a *quantitative* interpretation of separation logic [Batz et al. 2019]. Their logic can be used to lower bound the probability of a single outcome, making all but the private information retrieval example in this paper out of reach. While Fesefeldt et al. do support unbounded looping, the probability of nontermination is always added to the final expected values, and so it cannot be used to prove almost sure termination.

Fan et al. [2025] developed another logic based on Probabilistic Rely-Guarantee [McIver et al. 2016]. While it supports outcome splitting, the program must be re-instrumented to explicitly

declare where splitting occurs, whereas in pcOL splitting can be used anywhere. Fan et al. have a rule similar to Split2, but not Split1. In addition, their parallel composition rules do not give independence guarantees, and accordingly can only make conclusions about the local distributions of all threads, and not the *joint* distributions of all the variables in global memory, and programs must be proven to almost surely terminate externally to the logic.

Their logic is based on an oblivious adversary model—a weaker model than our unrestricted adversary. More programs are verifiable in the oblviious model, so it is sometimes preferred, however the logic of Fan et al. can only reason about obliviousness in limited ways, *i.e.*, by treating coin flip actions as atomic. Refer to Section 9.1 for more information about the connections between pcOL and oblivious schedulers.

CHAPTER 8

CASE STUDIES

To cap off the technical development, we now present three case studies to show how the cumulative features from the prior chapters come together for expressive verification of pcCL programs in pcOL. In general, the proofs require probabilistic reasoning from Chapter 4 to handle sampling operations; nondeterministic reasoning from Chapter 5 to handle observations about shared state; and concurrency reasoning from Chapter 7 to handle parallel composition of multiple threads.

8.1 THE ENTROPY MIXER

Entropy mixers produce pseudorandom bits by combining several potential sources of randomness. For example, the Linux kernel mixes entropy from physical sources such as disk timing and human interaction into its pseudorandom generator, which helps to ensure that an adversary cannot guess future random bits even if it learns the value of the seed [Gutterman et al. 2006]. A simple entropy mixer program is shown below.

$$
\begin{array}{l}
\textbf{EntropyMixer}: \\
\quad y \coloneqq 0 \fatsemi \\
\quad \left(\begin{array}{l} x_1 \coloneqq y \fatsemi \\ x_2 :\approx \mathbf{Ber}\left(\frac{1}{2}\right) \fatsemi \\ z \coloneqq \mathbf{xor}(x_1, x_2) \end{array} \;\middle\|\; y \coloneqq 1 \right)
\end{array}
$$

Here, x_2 is obtained via a fair coin flip and is therefore a high quality source of randomness. On the other hand, x_1 is a potentially unreliable source of entropy that it is obtained by observing the value of y from shared memory. We can make no guarantees about the way that x_1 is distributed, just as we can make no guarantees about physical phenomena such as disk timing. Despite that, we will prove that z, which is obtained by mixing x_1 and x_2 using an **xor**, is uniformly random.

We begin the proof by analyzing the two threads on their own. To do so, we first need a resource invariant, which will simply be $y \in \{0, 1\}$. The derivation for the first thread is shown

$$\dfrac{\dfrac{\dfrac{}{y \in \{0,1\} \vdash_{\mathsf{w}} \langle \lceil x_1 \mapsto Y * x_2 \mapsto X * \mathsf{own}(z) \rceil \rangle\ z := \mathbf{xor}(x_1, x_2)\ \langle \lceil z \mapsto \mathbf{xor}(Y, X) \rceil \rangle}\ \textsc{Assign}}{y \in \{0,1\} \vdash_{\mathsf{w}} \langle \bigoplus_{X \sim \mathbf{Ber}(\frac{1}{2})} \lceil x_1 \mapsto Y * x_2 \mapsto X * \mathsf{own}(z) \rceil \rangle\ z := \mathbf{xor}(x_1, x_2)\ \langle \bigoplus_{X \sim \mathbf{Ber}(\frac{1}{2})} \lceil z \mapsto \mathbf{xor}(Y, X) \rceil \rangle}\ \textsc{Split1}}{y \in \{0,1\} \vdash_{\mathsf{w}} \langle x_2 \sim \mathbf{Ber}\left(\tfrac{1}{2}\right) * \lceil x_1 \mapsto Y * \mathsf{own}(z) \rceil \rangle\ z := \mathbf{xor}(x_1, x_2)\ \langle z \sim \mathbf{Ber}\left(\tfrac{1}{2}\right) \rangle}\ \textsc{Consequence} \tag{8.1}$$

$$\dfrac{\dfrac{\dfrac{}{y \in \{0,1\} \vdash_{\mathsf{w}} \langle \lceil x_1 \mapsto Y * \mathsf{own}(x_2, z) \rceil \rangle\ x_2 :\approx \mathbf{Ber}\left(\tfrac{1}{2}\right)\ \langle x_2 \sim \mathbf{Ber}\left(\tfrac{1}{2}\right) * \lceil x_1 \mapsto Y * \mathsf{own}(z) \rceil \rangle}\ \textsc{Samp} \quad (8.1)}{y \in \{0,1\} \vdash_{\mathsf{w}} \langle \lceil x_1 \mapsto Y * \mathsf{own}(x_2, z) \rceil \rangle\ x_2 :\approx \mathbf{Ber}\left(\tfrac{1}{2}\right) \fatsemi z := \mathbf{xor}(x_1, x_2)\ \langle z \sim \mathbf{Ber}\left(\tfrac{1}{2}\right) \rangle}\ \textsc{Seq}}{y \in \{0,1\} \vdash_{\mathsf{w}} \langle \&_{Y \in \{0,1\}} \lceil x_1 \mapsto Y * \mathsf{own}(x_2, z) \rceil \rangle\ x_2 :\approx \mathbf{Ber}\left(\tfrac{1}{2}\right) \fatsemi z := \mathbf{xor}(x_1, x_2)\ \langle z \sim \mathbf{Ber}\left(\tfrac{1}{2}\right) \rangle}\ \textsc{NSplit2} \tag{8.2}$$

$$\dfrac{\dfrac{\dfrac{}{y \in \{0,1\} \vdash_{\mathsf{w}} \langle \lceil \mathsf{own}(x_1, x_2, z) \rceil \rangle\ x_1 := y\ \langle \&_{Y \in \{0,1\}} \lceil x_1 \mapsto Y * \mathsf{own}(x_2, z) \rceil \rangle}\ \textsc{AssignShared} \quad (8.2)}{y \in \{0,1\} \vdash_{\mathsf{w}} \langle \lceil \mathsf{own}(x_1, x_2, z) \rceil \rangle\ x_1 := y \fatsemi x_2 :\approx \mathbf{Ber}\left(\tfrac{1}{2}\right) \fatsemi z := \mathbf{xor}(x_1, x_2)\ \langle z \sim \mathbf{Ber}\left(\tfrac{1}{2}\right) \rangle}\ \textsc{Seq}}{y \in \{0,1\} \vdash \langle \lceil \mathsf{own}(x_1, x_2, z) \rceil \rangle\ x_1 := y \fatsemi x_2 :\approx \mathbf{Ber}\left(\tfrac{1}{2}\right) \fatsemi z := \mathbf{xor}(x_1, x_2)\ \langle z \sim \mathbf{Ber}\left(\tfrac{1}{2}\right) \rangle}\ \textsc{Strengthen} \tag{8.3}$$

Figure 8.1: Derivation for the first thread of the entropy mixer.

in Figure 8.1, and is best read from the lowermost precondition up to the uppermost triple, and then back down to the lowermost postcondition.

We start in (8.3) with the precondition $\lceil \mathsf{own}(x_1, x_2, z) \rceil$, simply stating that the thread has permission to read and write all of its local variables. Since the thread contains an observation from shared state, we will need to use a weak triple, but we are able to strengthen it at the last step. The first step is to break up the sequential composition and use AssignShared to determine that $\&_{Y \in \{0,1\}} \lceil x_1 \mapsto Y * \mathsf{own}(x_2, z) \rceil$ after assigning $x_1 := y$. The derivation for the second command in the sequence is shown in (8.2). The first step is to use NSplit2 to do case analysis on the possible nondeterministic values of x_1. Then, we can mechanically use Seq and Samp to determine that $x_2 \sim \mathbf{Ber}\left(\frac{1}{2}\right) * \lceil x_1 \mapsto Y * \mathsf{own}(z) \rceil$ for each value of Y.

Next, in (8.1), we do a second case analysis on the outcome of the coin flip using Split1 so that in the premise, the values of both x_1 and x_2 are deterministic, and therefore we can easily determine that the assignment $z := \mathbf{xor}(x_1, x_2)$ results in $\lceil z \mapsto (Y, X) \rceil$. In the conclusion of Split1, we get that $\bigoplus_{X \sim \mathbf{Ber}\left(\frac{1}{2}\right)} \lceil z \mapsto \mathbf{xor}(Y, X) \rceil$. Since Y is fixed outside the scope of the outcome conjunction and $\mathbf{xor}(Y, -)$ is a bijection, then we use a consequence to equivalently say that $z \sim \mathbf{Ber}\left(\frac{1}{2}\right)$, which no longer depends on Y, and therefore it carries over into the conclusion of NSplit2 and down the rest of the derivation. So, the first thread has the specification below.

$$y \in \{0,1\} \vdash \langle \lceil \mathsf{own}(x_1, x_2, z) \rceil \rangle\ x_1 := y \fatsemi x_2 :\approx \mathbf{Ber}\left(\tfrac{1}{2}\right) \fatsemi z := \mathbf{xor}(x_1, x_2)\ \langle z \sim \mathbf{Ber}\left(\tfrac{1}{2}\right) \rangle$$

Now, we move on to the second thread. Since the second thread has no local state, the precondition and postcondition will both simply be $\lceil \mathbf{true} \rceil$. The derivation is shown below.

$$\dfrac{\dfrac{\dfrac{}{\vdash \langle \lceil y \in \{0,1\} \rceil \rangle\ y := 1\ \langle \lceil y \mapsto 1 \rceil \rangle}\ \textsc{Assign}}{\vdash \langle \lceil \mathbf{true} \rceil * \lceil y \in \{0,1\} \rceil \rangle\ y := 1\ \langle \lceil \mathbf{true} \rceil * \lceil y \in \{0,1\} \rceil \rangle}\ \textsc{Consequence}}{y \in \{0,1\} \vdash \langle \lceil \mathbf{true} \rceil \rangle\ y := 1\ \langle \lceil \mathbf{true} \rceil \rangle}\ \textsc{Atom} \tag{8.4}$$

First we use Atom to move information about y into the triple. The assign rule tells us that $\lceil y \mapsto 1 \rceil$, which can be easily weakened to $\lceil y \in \{0,1\} \rceil$ using the rule of Consequence so that we can close the invariant.

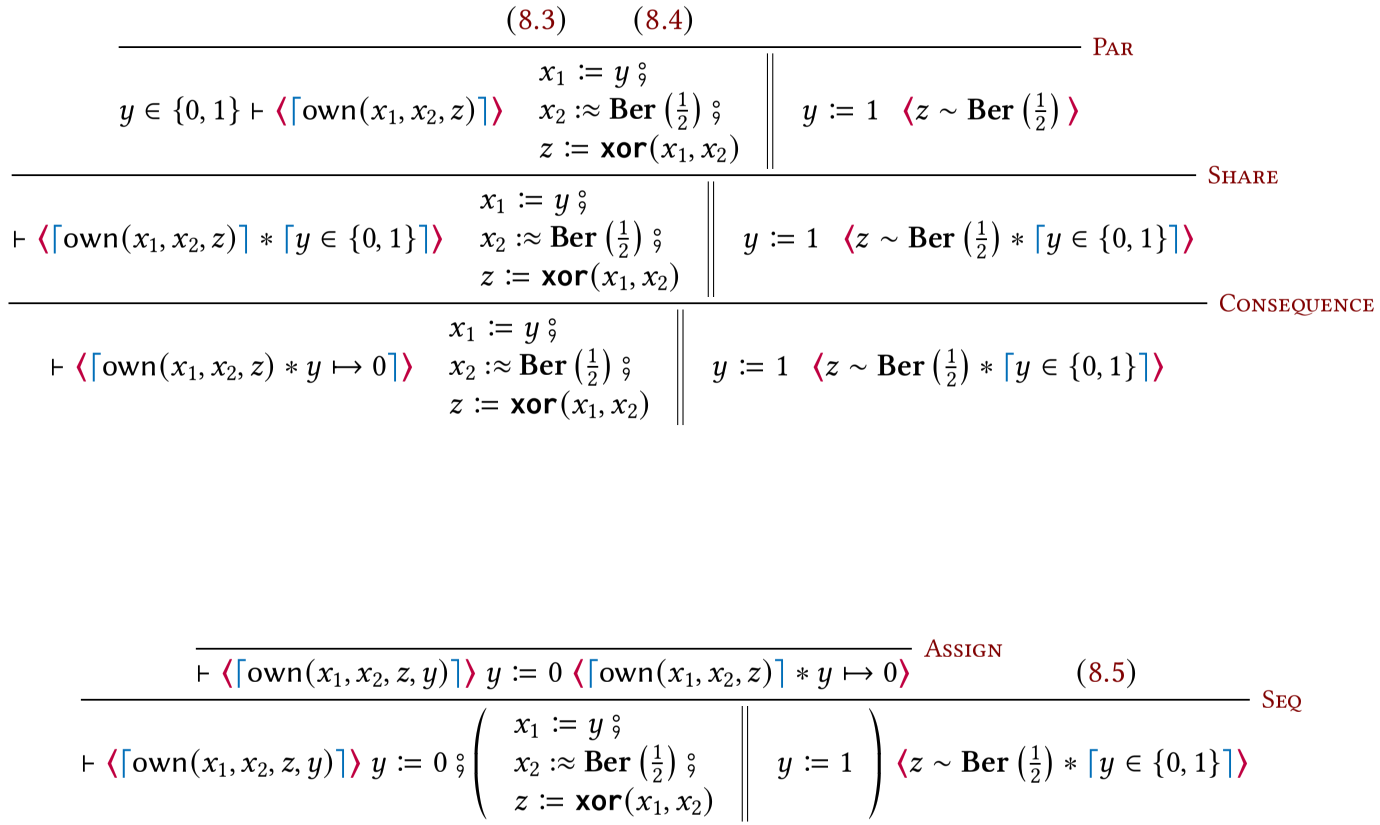


Figure 8.2: Entropy mixer main derivation.

Now, we derive a specification for the whole program in Figure 8.2. We begin by using Seq and Assign to determine that $\lceil y \mapsto 0 \rceil$ after $y \coloneqq 0$. Next, we come to the parallel composition in (8.5). First, we weaken the information about y to $\lceil y \in \{0,1\} \rceil$ and then we move the information about y into the invariant using Share. Finally, we simply use the Par rule with the prior derivations for the two threads to complete the proof, which results in the following triple, showing that z is distributed according to a fair coin flip, and is therefore a high quality source of entropy.

$$\vdash \langle \lceil \mathsf{own}(x_1, x_2, y, z) \rceil \rangle \ \texttt{EntropyMixer} \ \langle z \sim \mathbf{Ber}\left(\tfrac{1}{2}\right) * \lceil y \in \{0,1\} \rceil \rangle$$

Curiously, if we swap $x_1 :\approx \mathbf{Ber}\left(\frac{1}{2}\right)$ and $x_2 \coloneqq y$, then z is not a high quality source of entropy.

$$y \coloneqq 0 \ \fatsemi \left(\begin{array}{l} x_2 :\approx \mathbf{Ber}\left(\frac{1}{2}\right) \fatsemi \\ x_1 \coloneqq y \fatsemi \\ z \coloneqq \mathbf{xor}(x_1, x_2) \end{array} \ \middle\| \ y \coloneqq 1 \right)$$

In the entropy mixer variant above, the scheduler can use the outcome of the coin flip to decide how to schedule $y \coloneqq 1$ and thereby influence the value of z. For example, if $x_2 = 1$, then the scheduler can execute $y \coloneqq 1$ *before* $x_1 \coloneqq y$, therefore $x_1 = x_2 = 1$ and $z = \mathbf{xor}(1,1) = 0$. And if $x_2 = 0$, then the scheduler can execute $x_1 \coloneqq y$ before $y \coloneqq 1$, therefore $x_1 = x_2 = 0$ and $z = \mathbf{xor}(0,0) = 0$. In that case, we have $\lceil z \mapsto 0 \rceil$, *i.e.,* z is deterministically determined by the scheduler and is certainly not a high quality source of entropy.

The correctness proof in Figure 8.1 relied on applying NSplit2 before Split1, so that the nondeterministic value of x_1 is fixed before performing the coin flip and we could therefore conclude that x_2 is probabilistically independent of x_1. Swapping the order means that we would need to apply Split1 first to fix the value of x_1, but since x_2 is demonically chosen, fixing x_1 does not help us in ensuring that we always get a uniform distribution over z.

It may be surprising that the program becomes incorrect after swapping $x_1 :\approx \mathbf{Ber}\left(\frac{1}{2}\right)$ and $x_2 \coloneqq y$, two commands that have no data dependencies. Indeed, many compilers may perform that swap in reality, implying that it should not change the program behavior. In Section 9.1, we will discuss alternative semantic models that are more accommodating to command reordering.

ConcurrentShuffle :
  Partition ⨟ (**Shuffle**$_1$ ∥ **Shuffle**$_2$) ⨟ $a \coloneqq a_1 +\!\!+ a_2$

Partition :
  $a_1 \coloneqq [\,]$ ⨟ $a_2 \coloneqq [\,]$ ⨟ $i \coloneqq 0$ ⨟
  while $i < \mathbf{len}(a)$ **do**
    $(a_1 \coloneqq \mathbf{snoc}(a_1, a[i])) \oplus_{\frac{1}{2}} (a_2 \coloneqq \mathbf{snoc}(a_2, a[i]))$ ⨟
    $i \coloneqq i + 1$

Shuffle$_k$:
  $i_k \coloneqq \mathbf{len}(a_k)$ ⨟
  while $i_k > 0$ **do**
    $j_k :\approx \mathbf{unif}\,([0, .., i_k])$ ⨟
    $a_k \coloneqq \mathbf{swap}(a_k, i_k, j_k)$ ⨟
    $i_k \coloneqq i_k - 1$

Figure 8.3: A concurrent shuffling algorithm.

8.2 CONCURRENT SHUFFLE

Shuffling is a fundamental tool in randomized algorithms, where it is used used to prevent adversarial input orderings from forcing worse case runtime behavior [Motwani and Raghavan 1995] and to improve the convergence properties of machine learning algorithms [Gürbüzbalaban et al. 2021]. Bacher et al. [2018] showed that shuffling algorithms can be made up to seven times faster through parallelization, so we will now prove the correctness of a concurrent shuffling algorithm.

The program, shown in Figure 8.3 is inspired by the Rao-Sandelius shuffling algorithm [Rao 1961; Sandelius 1962]. Let $\ell_1 +\!\!+ \ell_2$ be an operator that concatenates two lists, $\mathbf{snoc}(\ell, x) \triangleq \ell +\!\!+ [x]$ be an operator that appends x to the end of ℓ (the reverse of **cons**), and $\mathbf{swap}(\ell, i, j)$ be an operator that return the list ℓ with the elements at positions i and j swapped.

The input list is first partitioned into two segments a_1 and a_2 using the **Partition** routine, which randomly appends each element to a_1 or a_2. Next, the two segments are shuffled in parallel using **Shuffle**$_k$, which is an implementation of the Fisher and Yates [1938] algorithm. Finally, the two shuffled segments are concatenated together to produce the output.[1] We will now prove that the output is a uniform permutation of the input.

[1]Note that although pcCL does not include functions or subroutines, **Partition** and **Shuffle**$_k$ are simply definitions of programs that are directly inserted into the body of **ConcurrentShuffle**, we only factor them into subroutines for ease of reading.

8.2.1 The Fisher-Yates Shuffle

Let us begin by verifying $\textbf{Shuffle}_k$, which is an implementation of the Fisher-Yates shuffling algorithm. The goal is to show that if a_k initially has some value given by the expression E, then a_k will hold a uniform permutation of E upon termination. For some list ℓ, let $\Pi(\ell)$ be the set of all permutations of ℓ. For the purposes of this example, we will presume that all lists do not contain duplicate values. The specification for $\textbf{Shuffle}_k$ is given below, and proven in Figure 8.4.

$$\vdash \langle \lceil a_k \mapsto E * \mathsf{own}(i_k, j_k) \rceil \rangle\ \textbf{Shuffle}_k\ \langle a_k \sim \textbf{unif}\,(\Pi(E))\ \rangle$$

To define the loop invariant, we first recursively define $\textbf{swaps}(n, \ell)$, which gives the set of possible lists obtained when $i_k \mapsto n$. Once i_k reaches n, all permutations of the elements above position n are accounted for, therefore $\textbf{swaps}(0, \ell) = \Pi(\ell)$ (all permutations of ℓ).

$$\textbf{swaps}(n, \ell) \triangleq \begin{cases} \{\ell\} & \text{if } n = \textbf{len}(\ell) - 1 \\ \{\textbf{swap}(\ell', n+1, j) \mid \ell' \in \textbf{swaps}(n+1, \ell), j \in \{0, .., n+1\}\} & \text{if } 0 \le n < \textbf{len}(\ell) - 1 \end{cases}$$

We now define the loop invariant φ using $\textbf{swaps}$. The rank R is given by i_k, which is bounded between 0 and $\textbf{len}(E) - 1$.

$$\varphi \triangleq \lceil i_k \mapsto R * 0 \le R < \textbf{len}(E) * \mathsf{own}(j_k) \rceil * (a_k \sim \textbf{unif}\,(\textbf{swaps}(R, E)))$$

The final postcondition $\varphi[0/R]$ can also be simplified as follows, since $\textbf{swaps}(0, E) = \Pi(E)$.

$$\varphi[0/R] = \lceil i_k \mapsto 0 * \mathsf{own}(j_k) \rceil * (a_k \sim \textbf{unif}\,(\Pi(E)))$$

The derivation for the loop body is shown in (8.6) of Figure 8.4 as a decorated program. When entering the loop body, we know that the tail of a_k is already shuffled. After performing the swap of i_k and j_k, we extend the shuffled tail by one position. Clearly, the shuffles remain uniformly distributed, since X is uniformly distributed, and the element at position N is also chosen uniformly. The remaining derivation of $\textbf{Shuffle}_k$ is quite mechanical given the derivation of the loop body, and is shown in (8.7) of Figure 8.4.

$$
\begin{array}{ll}
 & \langle \varphi * \lceil R = N > 0 \rceil \rangle \\
\textsc{Consequence} & \langle \lceil i_k \mapsto N * 0 < N < \mathbf{len}(E) * \mathrm{own}(j_k) \rceil * (a_k \sim \mathbf{unif}\,(\mathbf{swaps}(N, E))) \rangle \\
\textsc{Samp} & \quad j_k :\approx \mathbf{unif}\,([0, \ldots, i_k])\ ; \\
 & \langle \lceil i_k \mapsto N * 0 < N < \mathbf{len}(A) \rceil * (a_k \sim \mathbf{unif}\,(\mathbf{swaps}(N, E))) * (j_k \sim \mathbf{unif}\,([0, \ldots, N]) \rangle \\
\textsc{Consequence} & \langle \bigoplus_{X \sim \mathbf{unif}(\mathbf{swaps}(N,E))} \bigoplus_{Y \sim \mathbf{unif}([0,\ldots,N])} \lceil i_k \mapsto N * 0 < N < \mathbf{len}(E) * a_k \mapsto X * j_k \mapsto Y \rceil \rangle \\
\textsc{Split1}, \textsc{Assign} & \quad a_k := \mathbf{swap}(a_k, i_k, j_k)\ ; \\
 & \langle \bigoplus_{X \sim \mathbf{unif}(\mathbf{swaps}(N,E))} \bigoplus_{Y \sim \mathbf{unif}([0,\ldots,N])} \lceil i_k \mapsto N * 0 < N < \mathbf{len}(E) * a_k \mapsto \mathbf{swap}(X, N, Y) \rceil \rangle \\
\textsc{Consequence} & \langle \lceil i_k \mapsto N * 0 < N < \mathbf{len}(E) * \mathrm{own}(j_k) \rceil * (a_k \sim \mathbf{unif}\,(\mathbf{swaps}(N-1, E))) \rangle \\
\textsc{Assign} & \quad i_k := i_k - 1 \\
 & \langle \lceil i_k \mapsto N - 1 * 0 < N \leq \mathbf{len}(E) * \mathrm{own}(j_k) \rceil * (a_k \sim \mathbf{unif}\,(\mathbf{swaps}(N-1, E))) \rangle \\
\textsc{Consequence} & \langle (\&_{R=0}^{N-1}\, \varphi) \oplus_1 (\&_{R=N}^{\mathbf{len}(E)-1}\, \varphi) \rangle
\end{array}
\tag{8.6}
$$

$$
\dfrac{
\dfrac{}{\vdash \langle \lceil a_k \mapsto E * \mathrm{own}(i_k, j_k) \rceil \rangle\ i_k := \mathbf{len}(a_k) - 1\ \langle \lceil a_k \mapsto E * i_k \mapsto \mathbf{len}(E) - 1 * \mathrm{own}(j_k) \rceil \rangle}\ \textsc{Assign}
\qquad
\dfrac{
\dfrac{
\dfrac{(8.6)}{\vdash \langle\, \&_{R=0}^{\mathbf{len}(E)-1}\, \varphi \rangle\ \mathbf{while}\ i_k > 0\ \mathbf{do}\ (\cdots)\ \langle (a_k \sim \mathbf{unif}\,(\Pi(E))) * \lceil i_k \mapsto 0 * \mathrm{own}(j_k) \rceil \rangle}\ \textsc{BoundedRank}
}{\vdash \langle \lceil a_k \mapsto E * i_k \mapsto \mathbf{len}(EA) - 1 * \mathrm{own}(j_k) \rceil \rangle\ \mathbf{while}\ i_k > 0\ \mathbf{do}\ (\cdots)\ \langle a_k \sim \mathbf{unif}\,(\Pi(E)) \rangle}\ \textsc{Consequence}
}{\vdash \langle \lceil a_k \mapsto E * \mathrm{own}(i_k, j_k) \rceil \rangle\ i_k := \mathbf{len}(a_k) - 1\ ;\ \mathbf{while}\ i_k > 0\ \mathbf{do}\ (\cdots)\ \langle a_k \sim \mathbf{unif}\,(\Pi(E)) \rangle}\ \textsc{Seq}
\tag{8.7}
$$

Figure 8.4: Correctness proof for the $\mathbf{Shuffle}_k$ routine.

8.2.2 Partition

We now move on to prove correctness of the **Partition** routine. The program iterates over the input list a and assigns each element to a uniformly random binary bucket, indexed by 0 or 1. We can therefore index the outcomes of **Partition** using a bitstring $x \in \{0,1\}^{\mathbf{len}(A)}$ of length $\mathbf{len}(A)$ (where $a \mapsto A$). Our specification will show that those outcomes are uniformly distributed.

To formalize the specification, let us first define some notation. For some list ℓ and bitstring x, let $\ell[x]$ be the list obtained by filtering ℓ to only contain the indices i such that $x[i] = 1$, for example, $[1,2,3,4][1001] = [1,4]$. If x is shorter than ℓ, then we pad x on the right with zeros so that no additional items are taken, for example $[1,2,3,4][101] = [1,2,3,4][1010] = [1,3]$. In addition, let $\neg x$ be the bitwise negation of a bitstring, for example, $\neg 1001 = \neg 0110$. Given the new operators, we can state the specification for **Partition**:

$$\begin{array}{ll} & \langle \lceil a \mapsto A * \mathsf{own}(a_1, a_2, i, b, i_1, i_2, j_1, j_2) \rceil \rangle \\ \vdash & \quad \mathbf{Partition} \\ & \langle \lceil \mathsf{own}(a, i_1, i_2, j_1, j_2) \rceil * \bigoplus_{X \sim \mathbf{unif}\left(\{0,1\}^{\mathbf{len}(A)}\right)} \lceil a_1 \mapsto A[X] * a_2 \mapsto A[\neg X] \rceil \rangle \end{array} \tag{8.9}$$

After executing the program, the elements of a are split between a_1 and a_2 with the partition being represented by a uniformly chosen bitstring $x \in \{0,1\}^{\mathbf{len}(A)}$. To prove the specification, we first need to devise a loop invariant ψ, which is shown below.

$$\psi = \bigoplus_{X \sim \mathbf{unif}\left(\{0,1\}^{\mathbf{len}(A)-R}\right)} \lceil a \mapsto A * a_1 \mapsto A[X] * a_2 \mapsto A[\neg X] * i \mapsto \mathbf{len}(A) - R * 0 \le R \le \mathbf{len}(A) \rceil$$

The rank R is given by $\mathbf{len}(a) - i$, so that as i iterates from 0 to the end of the input list a, the loop deterministically moves closer to termination. On iteration i, the outcomes are indexed by a uniformly random bitstring $X \sim \mathbf{unif}\left(\{0,1\}^i\right)$ of length i. Each X corresponds to the first i elements of the input list a being allocated to a_1 and a_2 in a different way.

We now show the derivation of the loop body as a decorated program in Figure 8.5. We start by using Split1 to condition on the allocation of a from the previous iterations. The probabilistic choice to either extend a_1 or a_2 results in two fresh outcomes. We represent those outcomes by

$$
\begin{array}{l}
\langle \psi * [R = N > 0] \rangle \\
\left\langle \bigoplus_{X \sim \mathbf{unif}\left(\{0,1\}^{\mathbf{len}(A)-N}\right)} \lceil a \mapsto A * a_1 \mapsto A[X] * a_2 \mapsto A[\neg X] * i \mapsto \mathbf{len}(A) - N * 0 < N \leq \mathbf{len}(A) \rceil \right\rangle \\
\left\{ \begin{array}{ll}
\textsc{Split1,Frame} & \langle \lceil a \mapsto A * a_1 \mapsto A[X] * a_2 \mapsto A[\neg X] * i \mapsto \mathbf{len}(A) - N \rceil \rangle \\
\textsc{Prob} & \left(\begin{array}{ll} \textsc{Assign} & \langle \lceil a \mapsto A * a_1 \mapsto A[X] * a_2 \mapsto A[\neg X] * i \mapsto \mathbf{len}(A) - N \rceil \rangle \\ & \quad a_1 \coloneqq \mathbf{snoc}(a_1, a[i]) \\ & \langle \lceil a \mapsto A * a_1 \mapsto A[\mathbf{snoc}(X,1)] * a_2 \mapsto A[\neg \mathbf{snoc}(X,1)] * i \mapsto \mathbf{len}(A) - N \rceil \rangle \end{array} \right) \\
& \oplus_{\frac{1}{2}} \\
& \left(\begin{array}{ll} \textsc{Assign} & \langle \lceil a \mapsto A * a_1 \mapsto A[X] * a_2 \mapsto A[\neg X] * i \mapsto \mathbf{len}(A) - N \rceil \rangle \\ & \quad a_2 \coloneqq \mathbf{snoc}(a_2, a[i]) \\ & \langle \lceil a \mapsto A * a_1 \mapsto A[\mathbf{snoc}(X,0)] * a_2 \mapsto A[\neg \mathbf{snoc}(X,0)] * i \mapsto \mathbf{len}(A) - N \rceil \rangle \end{array} \right) \\
& \left\langle \bigoplus_{Y \sim \mathbf{Ber}\left(\frac{1}{2}\right)} \lceil a \mapsto A * a_1 \mapsto A[\mathbf{snoc}(X,Y)] * a_2 \mapsto A[\neg \mathbf{snoc}(X,Y)] * i \mapsto \mathbf{len}(A) - N \rceil \right\rangle \\
\textsc{Assign} & \quad i \coloneqq i + 1 \\
& \left\langle \bigoplus_{Y \sim \mathbf{Ber}\left(\frac{1}{2}\right)} \lceil a \mapsto A * a_1 \mapsto A[\mathbf{snoc}(X,Y)] * a_2 \mapsto A[\neg \mathbf{snoc}(X,Y)] * i \mapsto \mathbf{len}(A) - (N-1) \rceil \right\rangle
\end{array} \right. \\
\left\langle \bigoplus_{X \sim \mathbf{unif}\left(\{0,1\}^{\mathbf{len}(A)-N}\right)} \bigoplus_{Y \sim \mathbf{Ber}\left(\frac{1}{2}\right)} \lceil a \mapsto A * a_1 \mapsto A[\mathbf{snoc}(X,Y)] * a_2 \mapsto A[\neg \mathbf{snoc}(X,Y)] * i \mapsto \mathbf{len}(A) - (N-1) * 0 < N \leq \mathbf{len}(A) \rceil \right\rangle \\
\left\langle \bigoplus_{X \sim \mathbf{unif}\left(\{0,1\}^{\mathbf{len}(A)-(N-1)}\right)} \lceil a \mapsto A * a_1 \mapsto A[X] * a_2 \mapsto A[\neg X] * i \mapsto \mathbf{len}(A) - (N-1) * 0 \leq (N-1) \leq \mathbf{len}(A) \rceil \right\rangle \\
\langle (\&_{R=0}^{N-1} \psi) \oplus_1 (\&_{R=N}^{\mathbf{len}(A)}) \rangle
\end{array}
\tag{8.8}
$$

Figure 8.5: Derivation for the loop body of **Partition**.

appending either a 1 or 0 to the end of X, which dictates where the element $a[i]$ will be included. Since X is used directly in a_1 and is negated in a_2, then appending a 1 to X has the effect of adding an element to a_1, whereas adding a 0 has the effect of adding it to a_2. With uniform postconditions for the two branches, we can use Prob to put that postcondition under an outcome conjunction.

After incrementing i and applying a Consequence, we again satisfy the invariant, but with the rank decreased to $N-1$. Note that when $R = \mathbf{len}(A)$, the invariant simply states that i is zero and a_1 and a_2 are both empty.

$$\begin{aligned}\psi[\mathbf{len}(A)/R] &= \bigoplus_{X \sim \mathbf{unif}(\{0,1\}^0)} \lceil a \mapsto A * a_1 \mapsto A[X] * a_2 \mapsto A[\neg X] * i \mapsto 0 \rceil \\ &= \lceil a \mapsto A * a_1 \mapsto [] * a_2 \mapsto [] * i \mapsto 0 \rceil\end{aligned}$$

That is precisely the postcondition that we get after $a_1 := [] \fatsemi a_2 := [] \fatsemi i := 0$, the opening sequence of **Partition**. In addition, when $R = 0$, we essentially get the desired postcondition of **Partition** from (8.9).

$$\begin{aligned}\psi[0/R] &= \bigoplus_{X \sim \mathbf{unif}(\{0,1\}^{\mathbf{len}(A)})} \lceil a \mapsto A * a_1 \mapsto A[X] * a_2 \mapsto A[\neg X] * i \mapsto \mathbf{len}(A) \rceil \\ &\Rightarrow \bigoplus_{X \sim \mathbf{unif}(\{0,1\}^{\mathbf{len}(A)})} \lceil a_1 \mapsto A[X] * a_2 \mapsto A[\neg X] * \mathsf{own}(a) \rceil\end{aligned}$$

Therefore, we can complete the derivation using an application of BoundedRank.

$$\frac{(8.8)}{\vdash \langle \psi[\mathbf{len}(A)/R] \rangle \ \mathbf{while}\ i < \mathbf{len}(a)\ \mathbf{do}\ (..)\ \langle \psi[0/R] \rangle}\ \textsc{BoundedRank}$$

8.2.3 The Main Algorithm

To complete the proof of **ConcurrentShuffle**, we now simply put the previous pieces together. The final specification simply states that if $a \mapsto A$ before the program runs, then a will be a uniform permutation of A after the program execution.

$$\vdash \langle \lceil a \mapsto A * \mathsf{own}(a_1, a_2, i, i_1, i_2, j_1, j_2) \rceil \rangle\ \mathbf{ConcurrentShuffle}\ \langle a \sim \mathbf{unif}\ (\Pi(A)) \rangle$$

$$\dfrac{\dfrac{\dfrac{(8.7)}{\vdash \langle \lceil a_1 \mapsto A[X] * \mathsf{own}(i_1, j_1)\rceil\rangle\ \mathbf{Shuffle}_1\ \langle a_1 \sim d_1\rangle} \qquad \dfrac{(8.7)}{\vdash \langle \lceil a_2 \mapsto A[\neg X] * \mathsf{own}(i_2, j_2)\rceil\rangle\ \mathbf{Shuffle}_1\ \langle a_2 \sim d_2\rangle}\ \textsc{Par}}{\vdash \langle \lceil a_1 \mapsto A[X] * \mathsf{own}(i_1, j_1)\rceil * \lceil a_2 \mapsto A[\neg X] * \mathsf{own}(i_2, j_2)\rceil\rangle\ \mathbf{Shuffle}_1 \parallel \mathbf{Shuffle}_2\ \langle a_1 \sim d_1 * a_2 \sim d_2\rangle}\ \textsc{Frame}}{\vdash \langle \lceil a_1 \mapsto A[X] * \mathsf{own}(i_1, j_1)\rceil * \lceil a_2 \mapsto A[\neg X] * \mathsf{own}(i_2, j_2)\rceil * \lceil \mathsf{own}(a)\rceil\rangle\ \mathbf{Shuffle}_1 \parallel \mathbf{Shuffle}_2\ \langle a_1 \sim d_1 * a_2 \sim d_2 * \lceil \mathsf{own}(a)\rceil\rangle}\ \textsc{Consequence}}{\vdash \langle \lceil a_1 \mapsto A[X] * a_2 \mapsto A[\neg X] * \mathsf{own}(..)\rceil\rangle\ \mathbf{Shuffle}_1 \parallel \mathbf{Shuffle}_2\ \langle \bigoplus_{A_1 \sim d_1} \bigoplus_{A_2 \sim d_2} \lceil a_1 \mapsto A_1 * a_2 \mapsto A_2 * \mathsf{own}(a)\rceil\rangle} \tag{8.10}$$

$$\dfrac{\dfrac{\dfrac{}{\vdash \langle \lceil a_1 \mapsto A_1 * a_1 \mapsto A_2 * \mathsf{own}(a)\rceil\rangle\ a := a_1 +\!\!+ a_2\ \langle \lceil a \mapsto A_1 +\!\!+ A_2\rceil\rangle}\ \textsc{Assign}}{\vdash \langle \bigoplus_{A_2 \sim d_2} \lceil a_1 \mapsto A_1 * a_1 \mapsto A_2 * \mathsf{own}(a)\rceil\rangle\ a := a_1 +\!\!+ a_2\ \langle \bigoplus_{A_2 \sim d_2} \lceil a \mapsto A_1 +\!\!+ A_2\rceil\rangle}\ \textsc{Split1}}{\vdash \langle \bigoplus_{A_1 \sim d_1} \bigoplus_{A_2 \sim d_2} \lceil a_1 \mapsto A_1 * a_1 \mapsto A_2 * \mathsf{own}(a)\rceil\rangle\ a := a_1 +\!\!+ a_2\ \langle \bigoplus_{A_1 \sim d_1} \bigoplus_{A_2 \sim d_2} \lceil a \mapsto A_1 +\!\!+ A_2\rceil\rangle}\ \textsc{Split1} \tag{8.11}$$

$$\dfrac{\dfrac{\dfrac{(8.10) \qquad (8.11)}{\vdash \langle \lceil a_1 \mapsto A[X] * a_2 \mapsto A[\neg X] * \mathsf{own}(..)\rceil\rangle\ (\mathbf{Shuffle}_1 \parallel \mathbf{Shuffle}_2) \fatsemi a := a_1 +\!\!+ a_2\ \langle \bigoplus_{A_1 \sim d_1} \bigoplus_{A_2 \sim d_2} \lceil a \mapsto A_1 +\!\!+ A_2\rceil\rangle}\ \textsc{Seq}}{\vdash \langle \bigoplus_{X \sim d} \lceil a_1 \mapsto A[X] * a_2 \mapsto A[\neg X] * \mathsf{own}(..)\rceil\rangle\ (\mathbf{Shuffle}_1 \parallel \mathbf{Shuffle}_2) \fatsemi a := a_1 +\!\!+ a_2\ \langle \bigoplus_{X \sim d} \bigoplus_{A_1 \sim d_1} \bigoplus_{A_2 \sim d_2} \lceil a \mapsto A_1 +\!\!+ A_2\rceil\rangle}\ \textsc{Split1}}{\vdash \langle \lceil \mathsf{own}(a, i_1, i_2, j_1, j_2)\rceil * \bigoplus_{X \sim d} \lceil a_1 \mapsto A[X] * a_2 \mapsto A[\neg X]\rceil\rangle\ (\mathbf{Shuffle}_1 \parallel \mathbf{Shuffle}_2) \fatsemi a := a_1 +\!\!+ a_2\ \langle a \sim \mathbf{unif}\,(\Pi(A))\rangle}\ \textsc{Consequence} \tag{8.12}$$

$$\dfrac{\dfrac{(8.9) \qquad (8.12)}{\vdash \langle \lceil a \mapsto A * \mathsf{own}(a_1, a_2, i, i_1, i_2, j_1, j_2)\rceil\rangle\ \mathbf{Partition} \fatsemi (\mathbf{Shuffle}_1 \parallel \mathbf{Shuffle}_2) \fatsemi a := a_1 +\!\!+ a_2\ \langle a \sim \mathbf{unif}\,(\Pi(A))\rangle}\ \textsc{Seq}$$

Figure 8.6: Main derivation for the concurrent shuffle.

The full derivation is shown in Figure 8.6. For brevity, we let:

$$d \triangleq \mathbf{unif}\left(\{0,1\}^{\mathtt{len}(A)}\right) \qquad d_1 \triangleq \mathbf{unif}\left(\Pi(A[X])\right) \qquad d_2 \triangleq \mathbf{unif}\left(\Pi(A[\neg X])\right)$$

The first step in the derivation is to apply the specification for **`Partition`** from Section 8.2.2, after which point the outcomes are indexed by a bitstring $X \in \{0,1\}^{\mathtt{len}(a)}$, and are distributed according to d. Each X witnesses the way that the elements of a are partitioned between a_1 and a_2. Moving to (8.12), the next step is to use Split1 to condition on X. Then, in (8.10), we apply the Par rule using the specification for $\mathbf{Shuffle}_k$ from Section 8.2.1 for each thread, which gives us that $a_1 \sim d_1 * a_2 \sim d_2$. That is a_1 and a_2 are independently and uniformly distributed permutations of their original values. Using Consequence and the distributive law of $\bigoplus$, we rewrite the postcondition to be $\bigoplus_{A_1 \sim d_1} \bigoplus_{A_2 \sim d_2} \lceil a_1 \mapsto A_1 * a_2 \mapsto A_2 \rceil$. That allows us to conclude that $\bigoplus_{A_1 \sim d_1} \bigoplus_{A_2 \sim d_2} \lceil a \mapsto A_1 +\!\!+ A_2 \rceil$ after executing $a \coloneqq a_1 +\!\!+ a_2$ in (8.11).

Returning to (8.12) in the conclusion of Split1, the final step is to prove that the three nested outcome conjunctions imply that a is a uniform permutation of A. Take any $\ell \in \Pi(A)$ and let $n = \mathtt{len}(A)$. For every $0 \leq k \leq n$, there is exactly one split X such that X assigns the elements $\ell[0], \ldots, \ell[k-1]$ to a_1 and the rest to a_2, and X occurs with probability $\frac{1}{2^n}$. Further, given that a was partitioned according to X, then a_1 and a_2 are shuffled so that $a_1 +\!\!+ a_2 = \ell$ with probability $\frac{1}{k!}\frac{1}{(n-k)!}$. Thus the probability of getting the permutation ℓ is $\sum_{k=0}^{n} \frac{1}{2^n}\frac{1}{k!}\frac{1}{(n-k)!}$, which is equal to $\frac{1}{n!}$ by a straightforward application of the Binomial Theorem. Since there are exactly $|\Pi(A)| = n!$ permutations, that means that each permutation is produced uniformly.

8.3 PRIVATE INFORMATION RETRIEVAL

Private information retrieval (PIR) protocols allow users to fetch data from a server without the database operator learning what data was requested [Chor et al. 1998]. A simple form of private information retrieval is modeled in the program shown in Figure 8.7. The $\mathbf{Fetch}_k$ programs process a bitstring query q_k, indicating which entries of the database d to return. Those entries are then bitwise xor'ed together to form the response r_k.

$$
\begin{array}{ll}
\textbf{PrivFetch}: & \textbf{Fetch}_k: \\
\quad q_1 :\approx \textbf{unif}\,(\{0,1\}^n)\,; & \quad i_k \coloneqq 0\,;\, r_k \coloneqq 0\,; \\
\quad q_2 \coloneqq \textbf{xor}(q_1, x)\,; & \quad \textbf{while}\ i_k < \textbf{len}(q_k)\ \textbf{do} \\
\quad \textbf{fetch}_1 \parallel \textbf{fetch}_2\,; & \qquad r_k \coloneqq \textbf{xor}(r_k, q_k[i_k] \cdot d[i_k])\,; \\
\quad r \coloneqq \textbf{xor}(r_1, r_2) & \qquad i_k \coloneqq i_k + 1
\end{array}
$$

Figure 8.7: A concurrent private information retrieval protocol.

Private retrieval is implemented in **PrivFetch**. The input x is a *one-hot* bit string of length n $\textbf{1hot}(K)$, with a 1 in position K (indicating the index of the data to retrieve) and zeros everywhere else. Two queries are then made concurrently. The first one uses a uniformly random bit string q_1, and the second uses the same random string xor'ed with $\textbf{1hot}(K)$.

The final result r is an xor of the two responses, which reveals the data at position K. Barthe et al. [2020] proved the correctness of a similar example in PSL, but their version was sequential; both fetches happened within a single for loop. Our version better models a distributed system where the computation does not occur in lockstep, but rather occurs concurrently. We first present a specification for $\textbf{Fetch}_k$, which states that r_k is an xor of data entries i, such that $q_k[i] = 1$, subject to the invariant that $d \mapsto D$.

$$
d \mapsto D \vdash \langle \lceil q_k \mapsto E \rceil * \lceil \mathsf{own}(i_k, r_k) \rceil \rangle\ \textbf{Fetch}_k\ \langle \lceil r_k \mapsto \mathop{\textbf{xor}}_{0 \le i < n : E[i]=1} D[i] \rceil \rangle
$$

The derivation of the specification is shown in Figure 8.8. We first use Seq and Assign in (8.15) to conclude that $i_k \mapsto 0$ and $r_k \mapsto 0$ after the initialization. The next step is to analyze the loop, which requires a loop invariant. The invariant φ simply states that after i_k iterations, r_k holds the xor of the first i_k elements of D, which were requested according to the query q_k.

$$
\varphi \triangleq \lceil q_k \mapsto E * i_k \mapsto n - R * 0 \le R \le n * r_k \mapsto \mathop{\textbf{xor}}_{0 \le i < n-R : E[i]=1} D[i] \rceil
$$

The rank R indicates the remaining number of iterations until termination, and it clearly bounded between 0 and n. We can also specialize φ to obtain the precondition and postcondition of the whole loop, shown below:

$$
\varphi[n/R] = \lceil q_k \mapsto E * i_k \mapsto 0 * r_k \mapsto 0 \rceil \qquad \varphi[0/R] = \lceil q_k \mapsto E * i_k \mapsto n * r_k \mapsto \mathop{\textbf{xor}}_{0 \le i < n : E[i]=1} D[i] \rceil
$$

$$\dfrac{\dfrac{\dfrac{}{\langle \lceil q_k \mapsto E * i_k \mapsto n - N * 0 < N \leq n * r_k \mapsto \underset{0 \leq i < n-N : E[i]=1}{\mathbf{xor}}\, D[i] \rceil * \lceil d \mapsto D \rceil \rangle}\ \textsc{Assign}}{\vdash \quad r_k := \mathbf{xor}(r_k, q_k[i_k] \cdot d[i_k]) \quad \langle \lceil q_k \mapsto E * i_k \mapsto n - N * 0 \leq N - 1 \leq n * r_k \mapsto \underset{0 \leq i < n-(N-1) : E[i]=1}{\mathbf{xor}}\, D[i] \rceil * \lceil d \mapsto D \rceil \rangle}\ \textsc{Atom}}{\langle \lceil q_k \mapsto E * i_k \mapsto n - N * 0 < N \leq n * r_k \mapsto \underset{0 \leq i < n-N : E[i]=1}{\mathbf{xor}}\, D[i] \rceil \rangle}$$

$$d \mapsto D \vdash \quad r_k := \mathbf{xor}(r_k, q_k[i_k] \cdot d[i_k]) \quad \langle \lceil q_k \mapsto E * i_k \mapsto n - N * 0 \leq N - 1 \leq n * r_k \mapsto \underset{0 \leq i < n-(N-1) : E[i]=1}{\mathbf{xor}}\, D[i] \rceil \rangle$$

$$\vdots$$

$$\dfrac{\dfrac{}{\langle \lceil q_k \mapsto E * i_k \mapsto n - N * 0 \leq N - 1 \leq n * r_k \mapsto \underset{0 \leq i < n-(N-1) : E[i]=1}{\mathbf{xor}}\, D[i] \rceil \rangle}\ \textsc{Assign}}{d \mapsto D \vdash \quad i_k := i_k + 1 \quad \langle \lceil q_k \mapsto E * i_k \mapsto n - (N-1) * 0 \leq N - 1 \leq n * r_k \mapsto \underset{0 \leq i < n-(N-1) : E[i]=1}{\mathbf{xor}}\, D[i] \rceil \rangle}$$

$$\dfrac{\dfrac{}{d \mapsto D \vdash \langle \lceil q_k \mapsto E * i_k \mapsto n - N * 0 < N \leq n * r_k \mapsto \underset{0 \leq i < n-N : E[i]=1}{\mathbf{xor}}\, D[i] \rceil \rangle\ r_k := \mathbf{xor}(r_k, q_k[i_k] \cdot d[i_k]) \fatsemi i_k := i_k + 1\ \langle \varphi[N-1/R] \rangle}\ \textsc{Seq}}{d \mapsto D \langle \varphi * \lceil R = N > 0 \rceil \rangle\ r_k := \mathbf{xor}(r_k, q_k[i_k] \cdot d[i_k]) \fatsemi i_k := i_k + 1\ \langle (\&_{R=0}^{N-1}\, \varphi) \oplus_1 (\&_{R=N}^{n}\, \varphi) \rangle}\ \textsc{Consequence} \tag{8.13}$$

$$\dfrac{\dfrac{(8.13)}{d \mapsto D \vdash \langle\, \&_{R=0}^{n}\, \varphi \rangle\ \mathbf{while}\ i_k < \mathbf{len}(q_k)\ \mathbf{do}\ (\cdots)\ \langle \varphi[0/R] \rangle}\ \textsc{BoundedRank}}{d \mapsto D \vdash \langle \lceil q_k \mapsto E * i_k \mapsto 0 * r_k \mapsto 0 \rceil \rangle\ \mathbf{while}\ i_k < \mathbf{len}(q_k)\ \mathbf{do}\ (\cdots)\ \langle \lceil r_k \mapsto \underset{0 \leq i < n : E[i]=1}{\mathbf{xor}}\, D[i] \rceil \rangle}\ \textsc{Consequence} \tag{8.14}$$

$$\dfrac{\dfrac{}{d \mapsto D \vdash \langle \lceil q_k \mapsto E * \mathsf{own}(i_k, r_k) \rceil \rangle\ i_k := 0\ \langle \lceil q_k \mapsto E * i_k \mapsto 0 * \mathsf{own}(r_k) \rceil \rangle}\ \textsc{Assign}}{\vdots}$$

$$\dfrac{\dfrac{\dfrac{}{d \mapsto D \vdash \langle \lceil q_k \mapsto E * i_k \mapsto 0 * \mathsf{own}(r_k) \rceil \rangle\ r_k := 0\ \langle \lceil q_k \mapsto E * i_k \mapsto 0 * r_k \mapsto 0 \rceil \rangle}\ \textsc{Assign} \quad (8.14)}{d \mapsto D \vdash \langle \lceil q_k \mapsto E * i_k \mapsto 0 * \mathsf{own}(r_k) \rceil \rangle\ r_k := 0 \fatsemi \mathbf{while}\ i_k < \mathbf{len}(q_k)\ \mathbf{do}\ (\cdots)\ \langle \lceil r_k \mapsto \underset{0 \leq i < n : E[i]=1}{\mathbf{xor}}\, D[i] \rceil \rangle}\ \textsc{Seq}}{d \mapsto D \vdash \langle \lceil q_k \mapsto E * \mathsf{own}(i_k, r_k) \rceil \rangle\ i_k := 0 \fatsemi r_k := 0 \fatsemi \mathbf{while}\ i_k < \mathbf{len}(q_k)\ \mathbf{do}\ (\cdots)\ \langle \lceil r_k \mapsto \underset{0 \leq i < n : E[i]=1}{\mathbf{xor}}\, D[i] \rceil \rangle}\ \textsc{Seq} \tag{8.15}$$

Figure 8.8: Correctness derivation for the $\mathbf{Fetch}_k$ procedure.

Analyzing the loop is simply a matter of applying BoundedRank in (8.14), leaving only the derivation for the loop body in (8.13). After breaking up the sequential composition, we use Atom to gain access to d in the first command. Since r_k is updated to an xor of itself with $q_k[i_k] \cdot d[i_k]$, then its value will only change if $q_k[i_k] \cdot d[i_k]$ is nonzero. That means that r_k will only change if $q_k[i_k]$ is nonzero, which means that the data value at position i_k was requested according to the query q_k. Therefore, the application of Assign simply extends the bounds of the xor operation. After incrementing i_k, the loop invariant φ holds again, but now with $R = N - 1$.

Now, we move on to the main procedure, whose specification is shown below. That is, if $x \mapsto \mathbf{1hot}(K)$ before execution, then $r \mapsto D[K]$ after execution, proving that the correct piece of data was fetched.

$$\vdash \langle \lceil x \mapsto \mathbf{1hot}(K) * d \mapsto D * \mathsf{own}(q_1, q_1, r_1, r_2, r, i_1, i_2) \rceil \rangle \ \mathbf{PrivFetch} \ \langle \lceil r \mapsto D[K] \rceil \rangle$$

The derivation is shown in Figure 8.9. We begin by using Share to move the information about the database d into the invariant. After sampling into q_1, we use Split2 to condition on the outcome of the query. After assigning q_2, we get $\lceil q_1 \mapsto Q \rceil * \lceil q_2 \mapsto \mathbf{xor}(Q, \mathbf{1hot}(K)) \rceil$. We can then apply the Par rule in (8.16) to analyze the concurrent fetches to get the following postcondition:

$$\lceil r_1 \mapsto \mathop{\mathbf{xor}}_{0 \le i < n : Q[i]=1} D[i] \rceil * \lceil r_2 \mapsto \mathop{\mathbf{xor}}_{0 \le i < n : \mathbf{xor}(Q, \mathbf{1hot}(K))[i]=1} D[i] \rceil$$

After the final assignment $r := \mathbf{xor}(r_1, r_2)$, we want to conclude that $\lceil r \mapsto D[K] \rceil$. To do, so, we use the fact that r_1 and r_2 differ only in the term at index K.

$$\begin{aligned} \mathbf{xor}(r_1, r_2) &= \mathbf{xor}\left(\mathop{\mathbf{xor}}_{0 \le i < n : Q[i]=1} D[i], \mathop{\mathbf{xor}}_{0 \le i < n : \mathbf{xor}(Q, \mathbf{1hot}(K))[i]=1} D[i] \right) \\ &= \mathop{\mathbf{xor}}_{i=0}^{n-1} \begin{cases} \mathbf{xor}(D[i], D[i]) & \text{if } Q[i] = \mathbf{xor}(Q, \mathbf{1hot}(K))[i] = 1 \\ D[i] & \text{if } Q[i] = \neg\, \mathbf{xor}(Q, \mathbf{1hot}(K))[i] \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

Since $\mathbf{xor}(D[i], D[i]) = 0$, we can combine the first and last cases.

$$= \mathop{\mathbf{xor}}_{i=0}^{n-1} \begin{cases} D[i] & \text{if } Q[i] = \neg\, \mathbf{xor}(Q, \mathbf{1hot}(K))[i] \\ 0 & \text{otherwise} \end{cases}$$

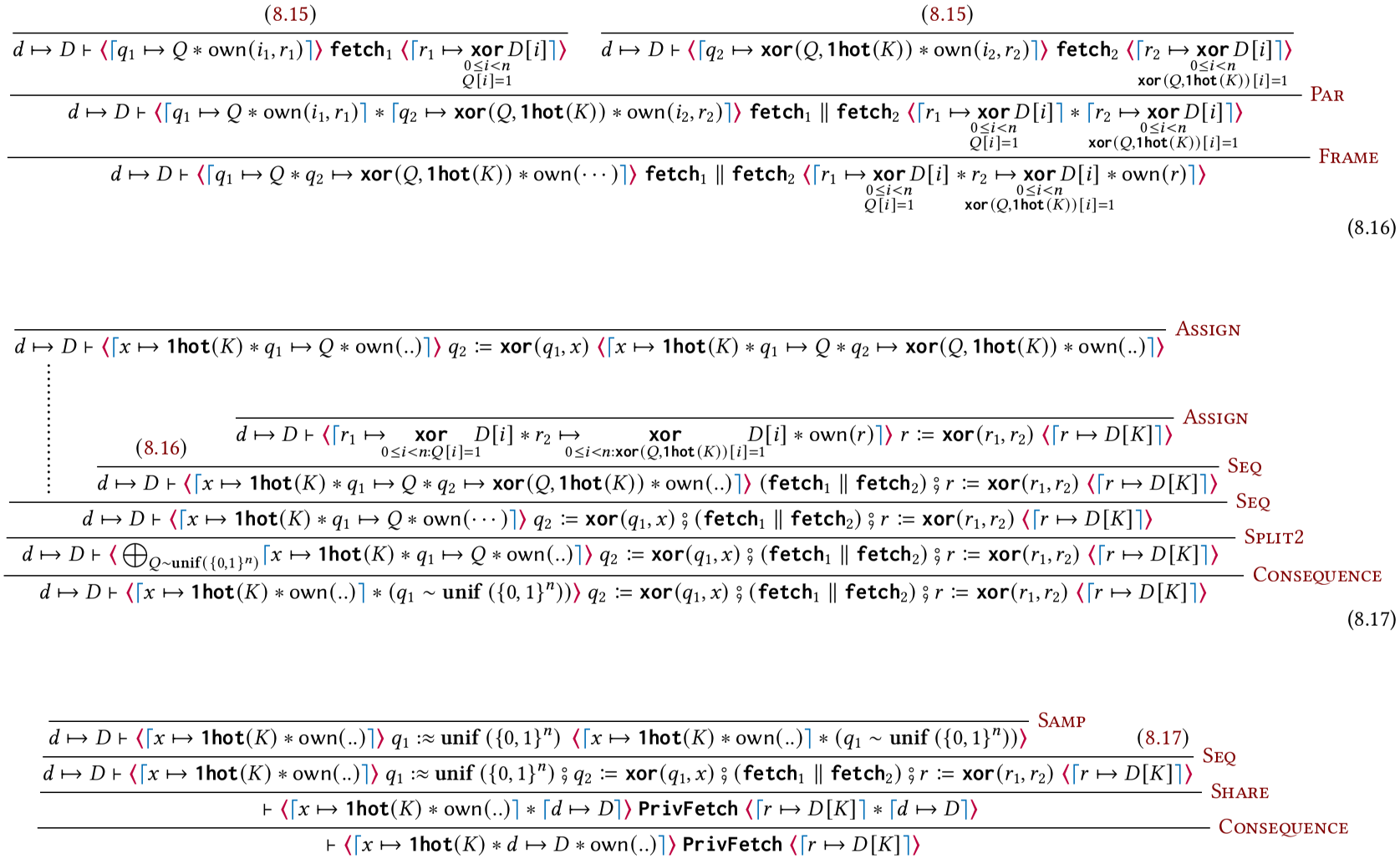


Figure 8.9: Correctness derivation for the **PrivFetch** program.

Note that $Q[i] = \neg\,\mathbf{xor}(Q, \mathbf{1hot}(K))[i]$ occurs only when $i = K$, so we can rewrite the conditional.

$$= \mathop{\mathbf{xor}}_{i=0}^{n-1} \begin{cases} D[i] & \text{if } i = K \\ 0 & \text{otherwise} \end{cases}$$

Since $\mathbf{xor}(x, 0) = x$, then we can drop all the zero terms.

$$= D[K]$$

Since the assertion $r \mapsto D[K]$ is convex and does not depend on Q, we meet the side conditions of the Split2 rule, and therefore the final postcondition is simply $\lceil r \mapsto D[K] \rceil$.

CHAPTER 9

CONCLUSION

Throughout the prior chapters, we have seen how reasoning about randomized program behavior (Chapters 3 and 4), nondeterministic program behavior (Chapter 5), and concurrency (Chapters 6 and 7), come together in pcOL. The result is an expressive logic, which can be used to compositionally prove correctness of concurrent programs in terms of the distribution of outcomes that they produce, as demonstrated in Chapter 8.

Whereas most prior approaches to formal methods for randomized concurrency have taken the approach of starting with an expressive concurrency logic and adding probabilistic reasoning [Li et al. 2025a; Tassarotti 2018; Tassarotti and Harper 2019], pcOL is designed from the ground up to support probabilistic reasoning natively. Consequently, pcOL has the expressivity to describe the distribution of outcomes that result from execution of the program, compositional reasoning principles based on probabilistic independence, and the ability to prove almost sure termination of unbounded loops.

However, just as Concurrent Separation Logic (CSL) [O'Hearn 2004; Brookes 2004] has been developed over the past two decades to support more expressive ways to reason about concurrency, there are also promising directions to further develop the base pcOL theory. We will discuss future directions in Section 9.1 as well as goals for mechanizing pcOL in Section 9.2 and applications in Section 9.3.

9.1 FURTHER THEORETICAL DEVELOPMENTS

Relaxed Scheduler Assumptions. The semantics of pcOL is based on the convex powerdomain, which represents a very strong scheduler model. Not only can the scheduler use randomization [Varacca and Winskel 2006], but it is also *adaptive* [Ben-David et al. 1990], in that it can adapt its decisions based on the random choices that occur during the program execution.

The adaptive model is too strong for some use cases. For example, some consensus algorithms

requires an *oblivious* scheduler, which cannot see the random values that have been sampled [Chor et al. 1994; Aspnes 2012]. At the end of Section 8.1, we also saw how the adaptive model is sensitive to reordering of commands that do not have data dependencies, which does not agree with the way that compilers may rewrite programs.

Operationally, we can think of an oblivious scheduler as a simple deterministic function $f\colon \{0,1\}^* \to \{0,1\}$, which produces its next binary choice given the finite sequence of prior choices that it has made. Denotationally, we can represent oblivious execution using a composition of the probability distribution monad and the state monad.

$$\mathcal{O}(X) \triangleq \{0,1\}^\omega \to \mathcal{D}(X \times \{0,1\}^\omega)$$

So, $\mathcal{O}(X)$ is a function whose input is an infinite stream of scheduler decisions ρ and whose output is a probability distribution over pairs (x, ρ'), where x is the result of the computation and ρ' is the suffix of ρ with the consumed scheduling bits removed from the beginning. We define a nondeterministic choice $o_1 \,\&\, o_2$ as a function that takes in a stream σ and returns the result of running o_1 (resp. o_2) if the first bit of σ is 1 (resp. 0).

$$(o_1 \,\&\, o_2)(\rho) \triangleq \begin{cases} o_1(\rho') & \text{if } \rho = 1 \cdot \rho' \\ o_2(\rho') & \text{if } \rho = 0 \cdot \rho' \end{cases}$$

The linearization function $\mathcal{L}$ for pomsets with formulae was originally formulated to be parametric over a computational domain D [Zilberstein et al. 2025a], so the underlying semantic model of PCOL could be immediately instantiated using $\mathcal{O}$ instead of $\mathcal{C}$. Letting $\mathcal{L}_D$ be the function from Section 6.4 instantiated to the domain D, and given any pomset $\boldsymbol{\alpha}$ and schedule $\rho \in \{0,1\}^\omega$, the oblivious scheduler model produces an outcome of the adaptive scheduler model.

$$\mathcal{L}_{\mathcal{O}}(\boldsymbol{\alpha})(\rho)(\sigma) \in \mathcal{L}_{\mathcal{C}}(\boldsymbol{\alpha})(\sigma)$$

That means that all of the inference rules of PCOL carry over to the more constrained oblivious case, giving us an expressive base logic to reason about randomized concurrent programs in the oblivious scheduler model.

Of course, it is also possible to add more powerful rules that leverage obliviousness. Fan et al. [2025] recently introduced such a logic, but the only rule pertaining to obliviousness was the *lazy coin* principle, which allows one to treat a probabilistic choice between two atomic actions $a_1 \oplus_e a_2$ as a single atomic program, meaning that the scheduler effectively cannot make any choices between the probabilistic choice and the execution of the resulting action. However, any nondeterministic choice that occurs after the lazy coin cannot be assumed to be oblivious in that logic, meaning that programs must be written in a very specific style in order for the logic to apply. That opens up major risks when translating programs into code, especially since most real languages do not include a $C_1 \oplus_e C_2$ operator.

By contrast, PCOL opens new possibilities for compositional reasoning about obliviousness in terms of *probabilistic independence*. For example, the triple below is valid in the oblivious (but not adaptive) model since an adaptive scheduler can introduce a correlation between x and y.

$$\langle \lceil \mathsf{own}(x, y) \rceil \rangle \; x :\approx \mathbf{Ber}\left(\tfrac{1}{2}\right) \fatsemi y \leftarrow \{0,1\} \; \langle x \sim \mathbf{Ber}\left(\tfrac{1}{2}\right) * \&_{Y \in \{0,1\}} \lceil y \mapsto Y \rceil \rangle$$

One might imagine that such a triple can be proven by changing NASSIGN to produce a strongly frame preserving triple. However, that would not be sound. The difficulty in designing oblivious rules is that information about random choices can still leak to the scheduler through side channels. For example, in the following program, the value of z might be decided by either the first or second bit in the scheduling sequence depending on the initial coin flip into x.

$$x :\approx \mathbf{flip}\left(\tfrac{1}{2}\right) \quad \fatsemi \quad \left(\begin{array}{l} \mathbf{if}\ x = 1\ \mathbf{then} \\ \quad y \leftarrow \{0,1\} \\ \mathbf{else} \\ \quad y := 2 \end{array} \right) \quad \fatsemi \quad z \leftarrow \{0,1\}$$

Therefore, using any sequence where the first two bits differ, the program above results in either $x = z$ with probability 1 or $x \neq z$ with probability 1, which is a clear correlation. So, rules like NASSIGN cannot *always* produce strong triples, but rather only when no random information has been leaked, making the sound formulation of such a rule challenging.

Fair Termination. Although pcOL can be used to prove almost sure termination, it is limited to cases where the behavior of parallel threads cannot affect the eventual probability of termination. Concurrent algorithms often require one thread to make progress before another thread can advance. As a very simple example, consider the program below where the first thread busy-waits until the second thread releases a resource.

$$\textbf{while}\ x = 0\ \textbf{do skip} \quad \| \quad x \coloneqq 1$$

The program above does *not* terminate in the semantic model from Chapter 6, since the scheduler can run the busy-wait forever and never allow the second thread to make progress. In other words, the scheduler is not *fair* [Francez 1986] because it can starve individual threads. Many randomized consensus, leader election, and synchronization algorithms only terminate with a fair scheduler [Ben-Or 1983; Lehmann and Rabin 1981; Itai and Rodeh 1990; Hart et al. 1983].

One difficulty in defining a fair semantics is that fairness is a non-compositional property of infinite traces; it is impossible to determine if an execution is fair by looking at its finite approximations. Fairness also results in unbounded nondeterminism [Apt and Plotkin 1986], as we can see in the following modification of the prior program, which now counts iterations.

$$i \coloneqq 0 \fatsemi \textbf{while}\ x = 0\ \textbf{do}\ i \coloneqq i + 1 \quad \| \quad x \coloneqq 1$$

Since the scheduler could delay the assignment $x \coloneqq 1$ for any finite number of steps, then i is effectively assigned to be a nondeterministic natural number at the end of the execution. That poses a problem because the convex powerdomain cannot represent unbounded nondeterministic choices [McIver and Morgan 2005]. One solution is to adopt the weaker notion of *probabilistic fairness*, which states that each nondeterministic choice must be made with probability at least $\varepsilon > 0$ [de Alfaro 1999]. Rather than standard fairness, which is a global property on infinite traces, probabilistic fairness is a local property of each nondeterministic choice. Another possibility is to move to a tree semantics, where unbounded nondeterminism is permissible [Back 1980, 1983].

As a further challenge, compositional approaches to liveness and fair termination verification are very complex even without randomization [D'Osualdo et al. 2021; Liang and Feng 2016, 2017;

Lee et al. 2025]. Moving to a probabilistic liveness setting, where each thread only makes progress probabilistically rather than deterministically, adds an additional layer of unpredictability. Enea et al. [2026] recently introduced a proof rule for reasoning about fair termination of randomized programs, but their approach only models round-based protocols rather than true concurrency.

As with the case of oblivious schedulers, fair schedulers are a restriction of the unconstrained schedulers of pcOL, meaning that the rules of pcOL over-approximate the behavior of programs under a fair scheduler model and therefore all the rules of pcOL apply to fair schedulers as is. The challenge is designing new rules for loops that allow us to prove termination of loops when termination depends on shared state.

Resources and Fine-Grained Concurrency. As with PSL [Barthe et al. 2020] and most other logics that build on it [Li et al. 2023; Bao et al. 2025, 2022, 2021], our resource model uses variables rather than dynamically allocated pointers. The challenge in adding dynamic allocation is that different heap addresses may be used in different random computation branches, and therefore the notion of separation becomes more difficult because the domains of the heaps in each distribution may not be disjoint. Outcome Separation Logic (OSL) combined dynamic allocation and separation with the ability to specify the probabilities of many outcomes [Zilberstein et al. 2024], but OSL used a simpler form of separation $\varphi \circledast \lceil P \rceil$, which could only combine outcome assertions φ with pure assertions $\lceil P \rceil$, and thus did not include any probabilistic independence guarantees.

Amaryllis extends PSL to support dynamic allocation and more general forms of *resource algebras* [Lohse et al. 2026], but it is limited to finitely supported distributions, and therefore the underlying model does not have a DCPO structure that can be used to give semantics to unbounded loops. Modern CSL implementations such as Iris [Jung et al. 2018, 2015] use resource algebras, so that additional types of physical and logical state can be added to govern the ways in which concurrent threads modify shared resources. Bluebell already includes permissions, which help to duplicate knowledge about read-only variables [Bao et al. 2025], however many other resources are used in practice making the Amaryllis approach more promising.

As an example, it seems reasonable that we should be able to prove the following triple. The program probabilistically increments x in two threads, and we want to conclude that x is uniformly distributed between 0 and 3 as a result.

$$\langle x \mapsto - \rangle\ x \coloneqq 0 \fatsemi \left(x \coloneqq x + 1 \oplus_{\frac{1}{2}} \mathbf{skip} \parallel x \coloneqq x + 2 \oplus_{\frac{1}{2}} \mathbf{skip} \right) \langle x \sim \mathbf{unif}\,([0,1,2,3]) \rangle$$

However, the triple above requires *ghost state* in order to describe how the shared state *evolves*, whereas a simple pcOL resource invariant can only describe a property that is *always* true.

An interesting alternative direction is to exploit *internal determinism* [Blelloch et al. 2012] to verify concurrent programs as if they were sequential. Moine et al. [2026] introduced the notion of *schedule-independent-safety* to show that a non-probabilistic concurrent program behaves equivalently in all interleavings, which allows them to sequentialize the program in any way they want using an angelic program logic. Using the same idea, the following rule that sequentializes a parallel composition would be sound in pcOL.

$$\frac{I \vdash \langle \varphi \rangle\ C_1 \fatsemi C_2\ \langle \psi \rangle \qquad \mathsf{ScheduleIndependent}(C_1 \parallel C_2)}{I \vdash \langle \varphi \rangle\ C_1 \parallel C_2\ \langle \psi \rangle}$$

As Moine et al. [2026] showed, for programs that are internally deterministic, sequentialization leads to simpler proofs as compared to using ghost state. Further, the proof of schedule independent safety can be fully automated. For the program above that results in $x \sim \mathbf{unif}\,([0,1,2,3])$, it would certainly be easy to verify the sequentialized version using pOL.

9.2 MECHANIZATION AND AUTOMATION

This dissertation presents the pcOL metatheory from the ground up. Although the traditional pen-and-paper approach best communicates the theory in a human readable way, applications of pcOL do not scale on paper. In Chapter 8, we already began to see the limits of typesetting pcOL proofs on a page. Going forward, it therefore makes sense to move the development of pcOL into a mechanized proof assistant to benefit from the machine-checked correctness guarantees, the scalability of working with code, and automation.

Mechanization of the pcOL metatheory is already underway in the Lean Theorem Prover [de Moura et al. 2015; Moura and Ullrich 2021]. Some key results such as the convex powerdomain[1], pomsets with formulae[2], and Lemma 7.8[3] have already been formalized. Lean is the ideal choice because much the underlying probability theory, domain theory, and topology needed to support pcOL are already formalized in mathlib [mathlib Community 2020], although more work is still needed to formalize all the foundational theories of pcOL, such as the convex powerdomain.

A further benefit of moving to Lean is the ability to take advantage of automation. For decades, program logics have been automated through the formulation of *weakest precondition* calculi [Dijkstra 1976], and that approach has extended to logics with outcome assertions too [Zhang et al. 2024]. However, weakest preconditions run into undecidable problems, such as finding loop invariants, for which automation is only possible via heuristic based approaches.

More recently, tools such as Harmonics' Aristotle [Achim et al. 2025] and Google's AlphaProof [Hubert et al. 2026] have shown that generative AI can be effectively tailored to solve complex mathematical problems in Lean. By the same token, tools like Aristotle and AlphaProof could be trained specifically to work in a logic like pcOL to prove the correctness of randomized concurrent programs, or at least to automate some parts of the proof.

Given that a growing amount of real-world software is being generated by opaque AI agents, rigorous semantic models to ensure that correctness proofs are faithful to our intuitions take on new importance. A specification $\langle \varphi \rangle\ C\ \langle \psi \rangle$ is only as meaningful as the definitions of the assertions φ and ψ, the program semantics, and the validity definition of the triple itself. So, well understood theories such as pcOL and pomsets with formulae will play a significant role towards software safety as AI use continues to proliferate. As we will see in the next section, many critical software components rely on the combination of concurrency and randomization, further motivating the need to mechanize pcOL.

[1] https://github.com/zilberstein/convex-powerset
[2] https://github.com/zilberstein/pomsets-with-formulae
[3] https://github.com/zilberstein/pcol-iris

9.3 APPLICATIONS

Randomized concurrent algorithms are crucial to modern distributed systems, but are also notoriously difficult to design and implement correctly. Formal methods offer a promising solution, but existing logics are not powerful enough to prove full functional correctness (especially liveness) of real programs. The developments of PCOL outlined in Sections 9.1 and 9.2 will unlock the ability verify programs in new domains. The following examples constitute only a brief survey of applications for PCOL. As a common thread, randomization is often instrumental to safety and performance of distributed systems.

Consensus Protocols. Consensus ensures that multiple independent processes agree on a single data value, despite failures, asynchrony, or conflicting proposals. It is fundamental to modern computing because it ensures consistency across distributed systems, underpinning technologies such as replicated databases, blockchain networks, and fault-tolerant services.

Fischer et al. [1985] famously showed that consensus cannot be achieved deterministically in an asynchronous system with faults. Hence, many classical approaches to consensus, such as Ben-Or's [1983] algorithm, fundamentally rely on randomization. In Ben-Or's algorithm, nodes broadcast their current values in an attempt to reach a majority vote, and decide on a value if enough proposals match. If consensus is not reached, nodes adopt a random value for the next round to probabilistically converge to a single value. Consensus therefore intersects with almost sure termination, as convergence is not guaranteed after a fixed number of rounds, but rather the probability of convergence goes to 1 as the system continues to execute. Similar algorithms have been applied to the related problem of leader election [Itai and Rodeh 1990].

More recently, consensus has attracted renewed interest in blockchain and cryptocurrency applications. Indeed, consensus is crucial to blockchain because it enables a decentralized network of untrusted participants to agree on a tamper-proof version of the truth without a central authority, thereby securing the network against fraud. New randomized consensus protocols

have been introduced as part of the Algorand [Gilad et al. 2017] and Avalanche [Amores-Sesar et al. 2024] blockchains, and Frank et al. [2025] advocated for the ongoing use of randomization in consensus to increase reliability and efficiency. Although formal methods have been applied to safety verification of blockchain consensus [Alturki et al. 2019; Esposito et al. 2025], liveness verification (*i.e.,* almost sure termination) is still out of reach.

Secure Multiparty Computation (MPC). MPC enables multiple parties to jointly compute a function over their private inputs without revealing anything beyond the final result. Randomization makes the protocol execution unpredictable, ensuring privacy and correctness by preventing adversaries from inferring private data or exploiting deterministic patterns.

MPC grew out of Shamir's [1979] secret sharing, which distributes randomized shares of secret data so that only authorized subsets of parties can reconstruct the original value. Randomness ensures that each individual fragment reveals nothing about the secret. Yao [1982] subsequently introduced the first secure two-party protocol for the millionaires' problem, where two millionaires determine which one of them is richer without disclosing their net worths. A general MPC algorithm was later developed by Goldreich et al. [1987].

Multiparty computation has been used in the real world to analyze income inequality [Lapets et al. 2018], to analyze Google advertisement conversions [Ion et al. 2020], and to decentralized the storage of cryptographic keys [Shoup 2000]. Refer to Lindell [2020] for a survey of the techniques and additional applications.

Sketching and Approximation. Compact probabilistic representations of large datasets enable efficient computations and queries in distributed and streaming environments. For example, Morris [1978] showed how to compactly represent large counts using randomized updates, and Flajolet [1985] extended the technique to count distinct elements in a data stream. That work formed the basis for the HyperLogLog algorithm [Flajolet et al. 2007], which is used in industrial database applications at Google [Heule et al. 2013], Meta [Honarkhah and Talebzadeh 2018], and

Amazon [Yu and Zarifis 2021]. Further examples of probabilistic data structures include Count-Min Sketch [Cormode and Muthukrishnan 2005], which is used for network monitoring by AT&T, and skip lists [Pugh 1990], which are used in database applications.

BIBLIOGRAPHY

Samson Abramsky and Achim Jung. 1995. *Domain Theory*. Oxford University Press, Inc., USA, 1–168.

Tudor Achim, Alex Best, Alberto Bietti, Kevin Der, Mathïs Fédérico, Sergei Gukov, Daniel Halpern-Leistner, Kirsten Henningsgard, Yury Kudryashov, Alexander Meiburg, Martin Michelsen, Riley Patterson, Eric Rodriguez, Laura Scharff, Vikram Shanker, Vladmir Sicca, Hari Sowrirajan, Aidan Swope, Matyas Tamas, Vlad Tenev, Jonathan Thomm, Harold Williams, and Lawrence Wu. 2025. Aristotle: IMO-level Automated Theorem Proving. arXiv:2510.01346 [cs.AI] https://arxiv.org/abs/2510.01346

Marcos Aguilera and Sam Toueg. 2012. Correctness Proof of Ben-Or's Randomized Consensus Algorithm. *Distributed Computing* 25 (10 2012). https://doi.org/10.1007/s00446-012-0162-z

Orestis Alpos and Christian Cachin. 2023. Do Not Trust in Numbers: Practical Distributed Cryptography with General Trust. In *Stabilization, Safety, and Security of Distributed Systems*, Shlomi Dolev and Baruch Schieber (Eds.). Springer Nature Switzerland, Cham, 536–551. https://doi.org/10.1007/978-3-031-44274-2_40

Musab A. Alturki, Jing Chen, Victor Luchangco, Brandon Moore, Karl Palmskog, Lucas Peña, and Grigore Roşu. 2019. Towards a Verified Model of the Algorand Consensus Protocol in Coq. In *Formal Methods. FM 2019 International Workshops: Porto, Portugal, October 7–11, 2019, Revised Selected Papers, Part I* (Porto, Portugal). Springer-Verlag, Berlin, Heidelberg, 362–367. https://doi.org/10.1007/978-3-030-54994-7_27

Ignacio Amores-Sesar, Christian Cachin, and Philipp Schneider. 2024. An Analysis of Avalanche Consensus. In *Structural Information and Communication Complexity*, Yuval Emek (Ed.). Springer Nature Switzerland, Cham, 27–44. https://doi.org/10.1007/978-3-031-60603-8_2

Dana Angluin. 1980. Local and global properties in networks of processors (Extended Abstract). In *Proceedings of the Twelfth Annual ACM Symposium on Theory of Computing* (Los Angeles, California, USA) *(STOC '80)*. Association for Computing Machinery, New York, NY, USA, 82–93. https://doi.org/10.1145/800141.804655

Krzysztof Apt. 1981. Ten Years of Hoare's Logic: A Survey–Part I. *ACM Trans. Program. Lang. Syst.* 3, 4 (oct 1981), 431–483. https://doi.org/10.1145/357146.357150

Krzysztof Apt and Gordon Plotkin. 1986. Countable nondeterminism and random assignment. *J. ACM* 33, 4 (aug 1986), 724–767. https://doi.org/10.1145/6490.6494

James Aspnes. 2012. Faster randomized consensus with an oblivious adversary. In *Proceedings of the 2012 ACM Symposium on Principles of Distributed Computing* (Madeira, Portugal) *(PODC '12)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/2332432.2332434

Steve Awodey. 2006. *Category Theory*. Oxford University Press. https://doi.org/10.1093/acprof:oso/9780198568612.001.0001

Axel Bacher, Olivier Bodini, Alexandros Hollender, and Jérémie O. Lumbroso. 2018. MergeShuffle: a very fast, parallel random permutation algorithm. In *Proceedings of the 11th International Conference on Random and Exhaustive Generation of Combinatorial Structures, GASCom 2018, Athens, Greece, June 18-20, 2018 (CEUR Workshop Proceedings, Vol. 2113)*, Luca Ferrari and Malvina Vamvakari (Eds.). CEUR-WS.org, 43–52. https://ceur-ws.org/Vol-2113/paper3.pdf

Ralph-Johan Back. 1980. Semantics of unbounded nondeterminism. In *Automata, Languages and Programming*, Jaco de Bakker and Jan van Leeuwen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 51–63. https://doi.org/10.1007/3-540-10003-2_59

Ralph-Johan Back. 1983. A continuous semantics for unbounded nondeterminism. *Theoretical Computer Science* 23, 2 (1983), 187–210. https://doi.org/10.1016/0304-3975(83)90055-5

Jialu Bao, Simon Docherty, Justin Hsu, and Alexandra Silva. 2021. A Bunched Logic for Conditional Independence. In

Proceedings of the 36th Annual ACM/IEEE Symposium on Logic in Computer Science (Rome, Italy) *(LICS '21)*. Association for Computing Machinery, New York, NY, USA, Article 13, 14 pages. https://doi.org/10.1109/LICS52264.2021.9470712

Jialu Bao, Emanuele D'Osualdo, and Azadeh Farzan. 2025. Bluebell: An Alliance of Relational Lifting and Independence for Probabilistic Reasoning. *Proc. ACM Program. Lang.* 9, POPL, Article 58 (Jan. 2025), 31 pages. https://doi.org/10.1145/3704894

Jialu Bao, Marco Gaboardi, Justin Hsu, and Joseph Tassarotti. 2022. A Separation Logic for Negative Dependence. *Proc. ACM Program. Lang.* 6, POPL, Article 57 (jan 2022), 29 pages. https://doi.org/10.1145/3498719

Gilles Barthe, Thomas Espitau, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2018. An Assertion-Based Program Logic for Probabilistic Programs. In *Programming Languages and Systems*. Springer International Publishing, Cham, 117–144. https://doi.org/10.1007/978-3-319-89884-1_5

Gilles Barthe, Thomas Espitau, Benjamin Grégoire, Justin Hsu, Léo Stefanesco, and Pierre-Yves Strub. 2015. Relational Reasoning via Probabilistic Coupling. In *Logic for Programming, Artificial Intelligence, and Reasoning*, Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 387–401. https://doi.org/10.1007/978-3-662-48899-7_27

Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. 2009. Formal certification of code-based cryptographic proofs. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Savannah, GA, USA) *(POPL '09)*. Association for Computing Machinery, New York, NY, USA, 90–101. https://doi.org/10.1145/1480881.1480894

Gilles Barthe, Justin Hsu, and Kevin Liao. 2020. A Probabilistic Separation Logic. *Proc. ACM Program. Lang.* 4, POPL, Article 55 (Jan. 2020), 30 pages. https://doi.org/10.1145/3371123

Kevin Batz, Adrian Gallus, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Tobias Winkler. 2022. Weighted Programming: A Programming Paradigm for Specifying Mathematical Models. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 66 (apr 2022), 30 pages. https://doi.org/10.1145/3527310

Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. 2021. Relatively complete verification of probabilistic programs: an expressive language for expectation-based reasoning. *Proc. ACM Program. Lang.* 5, POPL, Article 39 (jan 2021), 30 pages. https://doi.org/10.1145/3434320

Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Thomas Noll. 2019. Quantitative Separation Logic: A Logic for Reasoning about Probabilistic Pointer Programs. *Proc. ACM Program. Lang.* 3, POPL, Article 34 (Jan 2019), 29 pages. https://doi.org/10.1145/3290347

Jon Beck. 1969. Distributive laws. In *Seminar on Triples and Categorical Homology Theory*, B. Eckmann (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 119–140. https://doi.org/10.1007/BFb0083084

S. Ben-David, A. Borodin, R. Karp, G. Tardos, and A. Wigderson. 1990. On the power of randomization in online algorithms. In *Proceedings of the Twenty-Second Annual ACM Symposium on Theory of Computing* (Baltimore, Maryland, USA) *(STOC '90)*. Association for Computing Machinery, New York, NY, USA, 379–386. https://doi.org/10.1145/100216.100268

Michael Ben-Or. 1983. Another advantage of free choice: Completely asynchronous agreement protocols. In *Proceedings of the 2nd ACM Symposium on Principles of Distributed Computing* (Montreal, Quebec, Canada) *(PODC '83)*. Association for Computing Machinery, New York, NY, USA, 27–30. https://doi.org/10.1145/800221.806707

Lars Birkedal and Hongseok Yang. 2007. Relational Parametricity and Separation Logic. In *Foundations of Software Science and Computational Structures*. Springer Berlin Heidelberg, Berlin, Heidelberg, 93–107. https://doi.org/10.1007/978-3-540-71389-0_8

Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. 2012. Internally deterministic parallel

algorithms can be fast. *SIGPLAN Not.* 47, 8 (Feb. 2012), 181–192. https://doi.org/10.1145/2370036.2145840

Stephen Brookes. 2002. Traces, Pomsets, Fairness and Full Abstraction for Communicating Processes. In *CONCUR 2002 — Concurrency Theory*, Luboš Brim, Mojmír Křetínský, Antonín Kučera, and Petr Jančar (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 466–482. https://doi.org/10.1007/3-540-45694-5_31

Stephen Brookes. 2004. A Semantics for Concurrent Separation Logic. In *CONCUR 2004 - Concurrency Theory*, Philippa Gardner and Nobuko Yoshida (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 16–34. https://doi.org/10.1007/978-3-540-28644-8_2

Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving Fast with Software Verification. In *NASA Formal Methods*. Springer International Publishing, Cham, 3–11. https://doi.org/10.1007/978-3-319-17524-9_1

Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. 2009. Compositional Shape Analysis by Means of Bi-Abduction. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Savannah, GA, USA) *(POPL '09)*. Association for Computing Machinery, New York, NY, USA, 289–300. https://doi.org/10.1145/1480881.1480917

Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2011. Compositional Shape Analysis by Means of Bi-Abduction. *J. ACM* 58, 6, Article 26 (Dec 2011), 66 pages. https://doi.org/10.1145/2049697.2049700

Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. 2007. Local Action and Abstract Separation Logic. In *22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007)*. 366–378. https://doi.org/10.1109/LICS.2007.30

Quentin Carbonneaux, Noam Zilberstein, Christoph Klee, Peter W. O'Hearn, and Francesco Zappa Nardelli. 2022. Applying Formal Verification to Microkernel IPC at Meta. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Philadelphia, PA, USA) *(CPP 2022)*. Association for Computing Machinery, New York, NY, USA, 116–129. https://doi.org/10.1145/3497775.3503681

Aleksandar Chakarov and Sriram Sankaranarayanan. 2013. Probabilistic Program Analysis with Martingales. In *Computer Aided Verification*, Natasha Sharygina and Helmut Veith (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 511–526. https://doi.org/10.1007/978-3-642-39799-8_34

Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. 2007. Paxos made live: an engineering perspective. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing* (Portland, Oregon, USA) *(PODC '07)*. Association for Computing Machinery, New York, NY, USA, 398–407. https://doi.org/10.1145/1281100.1281103

Benny Chor, Amos Israeli, and Ming Li. 1994. Wait-Free Consensus Using Asynchronous Hardware. *SIAM J. Comput.* 23, 4 (1994), 701–712. https://doi.org/10.1137/S0097539790192635

Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. 1998. Private information retrieval. *J. ACM* 45, 6 (Nov. 1998), 965–981. https://doi.org/10.1145/293347.293350

Stephen A. Cook. 1978. Soundness and Completeness of an Axiom System for Program Verification. *SIAM J. Comput.* 7, 1 (feb 1978), 70–90. https://doi.org/10.1137/0207005

James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2012. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementa-*

tion (Hollywood, CA, USA) *(OSDI'12)*. USENIX Association, USA, 251–264. https://doi.org/10.5555/2387880.2387905

Ricardo Corin and Jerry den Hartog. 2006. A Probabilistic Hoare-style Logic for Game-Based Cryptographic Proofs. In *Automata, Languages and Programming*. Springer Berlin Heidelberg, Berlin, Heidelberg, 252–263. https://doi.org/10.1007/11787006_22

Graham Cormode and S. Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58–75. https://doi.org/10.1016/j.jalgor.2003.12.001

Luca de Alfaro. 1999. From Fairness to Chance. *Electronic Notes in Theoretical Computer Science* 22 (1999), 55–87. https://doi.org/10.1016/S1571-0661(05)80597-3

J. W. de Bakker and J. H. A. Warmerdam. 1990. Metric pomset semantics for a concurrent language with recursion. In *Semantics of Systems of Concurrent Processes*, Irène Guessarian (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 21–49. https://doi.org/10.1007/3-540-53479-2_2

Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description). In *Automated Deduction - CADE-25*, Amy P. Felty and Aart Middeldorp (Eds.). Springer International Publishing, Cham, 378–388. https://doi.org/10.1007/978-3-319-21401-6_26

Jerry den Hartog. 1999. Verifying Probabilistic Programs Using a Hoare like Logic. In *Advances in Computing Science — ASIAN'99*, P. S. Thiagarajan and Roland Yap (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 113–125. https://doi.org/10.1007/3-540-46674-6_11

Jerry den Hartog. 2002. *Probabilistic Extensions of Semantical Models*. Ph.D. Dissertation. Vrije Universiteit Amsterdam. https://core.ac.uk/reader/15452110

Edsger W. Dijkstra. 1971. Hierarchical ordering of sequential processes. *Acta Inf.* 1, 2 (June 1971), 115–138. https://doi.org/10.1007/BF00289519

Edsger W. Dijkstra. 1975. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM* 18, 8 (Aug 1975), 453–457. https://doi.org/10.1145/360933.360975

Edsger W. Dijkstra. 1976. *A Discipline of Programming*. Prentice-Hall. I–XVII, 1–217 pages.

Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. 2019. Scaling Static Analyses at Facebook. *Commun. ACM* 62, 8 (jul 2019), 62–70. https://doi.org/10.1145/3338112

Simon Docherty. 2019. *Bunched logics: a uniform approach*. Ph.D. Dissertation. University College London. https://discovery.ucl.ac.uk/id/eprint/10073115/

Emanuele D'Osualdo, Julian Sutherland, Azadeh Farzan, and Philippa Gardner. 2021. TaDA Live: Compositional Reasoning for Termination of Fine-grained Concurrent Programs. *ACM Trans. Program. Lang. Syst.* 43, 4, Article 16 (Nov. 2021), 134 pages. https://doi.org/10.1145/3477082

Luc Edixhoven and Sung-Shik Jongmans. 2022. Realisability of Branching Pomsets. In *Formal Aspects of Component Software*, Silvia Lizeth Tapia Tarifa and José Proença (Eds.). Springer International Publishing, Cham, 185–204. https://doi.org/10.1007/978-3-031-20872-0_11

Luc Edixhoven, Sung-Shik Jongmans, José Proença, and Ilaria Castellani. 2024. Branching pomsets: Design, expressiveness and applications to choreographies. *Journal of Logical and Algebraic Methods in Programming* 136 (2024), 100919. https://doi.org/10.1016/j.jlamp.2023.100919

Luc Edixhoven, Sung-Shik Jongmans, José Proença, and Guillermina Cledou. 2022. Branching Pomsets for Choreographies. *Electronic Proceedings in Theoretical Computer Science* 365 (Aug. 2022), 37–52. https://doi.org/10.4204/eptcs.365.3

Constantin Enea, Rupak Majumdar, Harshit Jitendra Motwani, and V. R. Sathiyanarayana. 2026. Verifying Almost-Sure Termination for Randomized Distributed Algorithms. *Proc. ACM Program. Lang.* 10, POPL, Article 49 (Jan. 2026), 30 pages. https://doi.org/10.1145/3776691

Andrea Esposito, Francesco P. Rossi, Marco Bernardo, Francesco Fabris, and Hubert Garavel. 2025. Formal Modeling and Verification of the Algorand Consensus Protocol in CADP. arXiv:2508.19452 [cs.DC] https://arxiv.org/abs/2508.19452

Weijie Fan, Hongjin Liang, Xinyu Feng, and Hanru Jiang. 2025. A Program Logic for Concurrent Randomized Programs in the Oblivious Adversary Model. In *Programming Languages and Systems*, Viktor Vafeiadis (Ed.). Springer Nature Switzerland, Cham, 322–348. https://doi.org/10.1007/978-3-031-91118-7_13

Ira Fesefeldt, Joost-Pieter Katoen, and Thomas Noll. 2022. Towards Concurrent Quantitative Separation Logic. In *33rd International Conference on Concurrency Theory (CONCUR 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 243)*, Bartek Klin, Sławomir Lasota, and Anca Muscholl (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 25:1–25:24. https://doi.org/10.4230/LIPIcs.CONCUR.2022.25

Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1985. Impossibility of distributed consensus with one faulty process. *J. ACM* 32, 2 (April 1985), 374–382. https://doi.org/10.1145/3149.214121

Ronald A. Fisher and Frank Yates. 1938. *Statistical Tables: for biological, agricultural and medical research* (4th ed. ed.). Oliver and Boyd, Edinburgh.

Philippe Flajolet. 1985. Approximate counting: A detailed analysis. *BIT* 25, 1 (March 1985), 113–134. https://doi.org/10.1007/BF01934993

Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. 2007. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. *Discrete Mathematics & Theoretical Computer Science* DMTCS Proceedings vol. AH, 2007 Conference on Analysis of Algorithms (AofA 07), Article 10 (Jan 2007). https://doi.org/10.46298/dmtcs.3545

Nissim Francez. 1986. *Fairness*. Springer, New York, NY. https://doi.org/10.1007/978-1-4612-4886-6

Reginald Frank, Octavio Lomeli, Neil Giridharan, Soujanya Ponnapalli, Marcos K. Aguilera, and Natacha Crooks. 2025. Real Life Is Uncertain. Consensus Should Be Too!. In *Proceedings of the 2025 Workshop on Hot Topics in Operating Systems* (Banff, AB, Canada) *(HotOS '25)*. Association for Computing Machinery, New York, NY, USA, 80–87. https://doi.org/10.1145/3713082.3730374

David Fremlin. 2001. *Measure Theory, Volume 2*.

Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. 2017. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 51–68. https://doi.org/10.1145/3132747.3132757

Michèle Giry. 1982. A categorical approach to probability theory. In *Categorical Aspects of Topology and Analysis*. Springer Berlin Heidelberg, Berlin, Heidelberg, 68–85. https://doi.org/10.1007/BFb0092872

Jay L. Gischer. 1988. The equational theory of pomsets. *Theoretical Computer Science* 61, 2 (1988), 199–224. https://doi.org/10.1016/0304-3975(88)90124-7

O. Goldreich, S. Micali, and A. Wigderson. 1987. How to play ANY mental game. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing* (New York, New York, USA) *(STOC '87)*. Association for Computing Machinery, New York, NY, USA, 218–229. https://doi.org/10.1145/28395.28420

Jan Grabowski. 1981. On Partial Languages. *Fundamenta Informaticae* 4, 2 (1981), 427–498. https://doi.org/10.3233/fi-1981-4210

Simon Oddershede Gregersen, Alejandro Aguirre, Philipp G. Haselwarter, Joseph Tassarotti, and Lars Birkedal. 2024a. Almost-Sure Termination by Guarded Refinement. *Proc. ACM Program. Lang.* 8, ICFP, Article 243 (Aug. 2024), 31 pages. https://doi.org/10.1145/3674632

Simon Oddershede Gregersen, Alejandro Aguirre, Philipp G. Haselwarter, Joseph Tassarotti, and Lars Birkedal. 2024b. Asynchronous Probabilistic Couplings in Higher-Order Separation Logic. *Proc. ACM Program. Lang.* 8, POPL, Article 26 (Jan. 2024), 32 pages. https://doi.org/10.1145/3632868

M. Gürbüzbalaban, A. Ozdaglar, and P. A. Parrilo. 2021. Why random reshuffling beats stochastic gradient descent. *Math. Program.* 186, 1–2 (March 2021), 49–84. https://doi.org/10.1007/s10107-019-01440-w

Zvi Gutterman, Benny Pinkas, and Tzachy Reinman. 2006. Analysis of the Linux Random Number Generator. Cryptology ePrint Archive, Paper 2006/086. https://eprint.iacr.org/2006/086

Sergiu Hart, Micha Sharir, and Amir Pnueli. 1983. Termination of Probabilistic Concurrent Programs. *ACM Trans. Program. Lang. Syst.* 5, 3 (July 1983), 356–380. https://doi.org/10.1145/2166.357214

Philipp G. Haselwarter, Kwing Hei Li, Alejandro Aguirre, Simon Oddershede Gregersen, Joseph Tassarotti, and Lars Birkedal. 2024. Approximate Relational Reasoning for Higher-Order Probabilistic Programs. arXiv:2407.14107 [cs.LO] https://arxiv.org/abs/2407.14107

Jifeng He, Karen Seidel, and Annabelle McIver. 1997. Probabilistic models for the guarded command language. *Science of Computer Programming* 28, 2 (1997), 171–192. https://doi.org/10.1016/S0167-6423(96)00019-6

Stefan Heule, Marc Nunkesser, and Alexander Hall. 2013. HyperLogLog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm. In *Proceedings of the 16th International Conference on Extending Database Technology* (Genoa, Italy) *(EDBT '13)*. Association for Computing Machinery, New York, NY, USA, 683–692. https://doi.org/10.1145/2452376.2452456

Charles Antony Richard Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580. https://doi.org/10.1145/363235.363259

Tony Hoare, Bernhard Möller, Georg Struth, and Ian Wehrman. 2011. Concurrent Kleene Algebra and its Foundations. *J. Log. Algebraic Methods Program.* 80, 6 (2011), 266–296. https://doi.org/10.1016/J.JLAP.2011.04.005

Mehrdad Honarkhah and Arya Talebzadeh. 2018. HyperLogLog in Presto: A Significantly Faster Way to Handle Cardinality Estimation. https://engineering.fb.com/2018/12/13/data-infrastructure/hyperloglog/. Engineering at Meta blog, accessed 2026.

Justin Hsu. 2017. *Probabilistic Couplings for Probabilistic Reasoning*. PhD dissertation. University of Pennsylvania. https://repository.upenn.edu/handle/20.500.14332/29996

Thomas Hubert, Rishi Mehta, Laurent Sartran, Miklós Z. Horváth, Goran Žužić, Eric Wieser, Aja Huang, Julian Schrittwieser, Yannick Schroecker, Hussain Masoom, Ottavia Bertolli, Tom Zahavy, Amol Mandhane, Jessica Yung, Iuliya Beloshapka, Borja Ibarz, Vivek Veeriah, Lei Yu, Oliver Nash, Paul Lezeau, Salvatore Mercuri, Calle Sönne, Bhavik Mehta, Alex Davies, Daniel Zheng, Fabian Pedregosa, Yin Li, Ingrid von Glehn, Mark Rowland, Samuel Albanie, Ameya Velingker, Simon Schmitt, Edward Lockhart, Edward Hughes, Henryk Michalewski, Nicolas Sonnerat, Demis Hassabis, Pushmeet Kohli, and David Silver. 2026. Olympiad-level formal mathematical reasoning with reinforcement learning. *Nature* 651 (2026), 607–613. https://doi.org/10.1038/s41586-025-09833-y

Mihaela Ion, Benjamin Kreuter, Erhan Nergiz, Sarvar Patel, Mariana Raykova, Shobhit Saxena, Karn Seth, David Shanahan, and Moti Yung. 2020. Private Intersection-Sum Protocols with Applications to Attributing Aggregate Ad Conversions. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*. 370–389. https://eprint.iacr.org/2019/723.pdf

Alon Itai and Michael Rodeh. 1990. Symmetry breaking in distributed networks. *Information and Computation* 88, 1 (1990), 60–87. https://doi.org/10.1016/0890-5401(90)90004-2

Bart Jacobs. 2008. Coalgebraic Trace Semantics for Combined Possibilitistic and Probabilistic Systems. *Electronic Notes in Theoretical Computer Science* 203, 5 (2008), 131–152. https://doi.org/10.1016/j.entcs.2008.05.023

Bart Jacobs. 2021. From Multisets over Distributions to Distributions over Multisets. In *Proceedings of the 36th Annual ACM/IEEE Symposium on Logic in Computer Science* (Rome, Italy) *(LICS '21)*. Association for Computing Machinery, New York, NY, USA, Article 39, 13 pages. https://doi.org/10.1109/LICS52264.2021.9470678

Radha Jagadeesan, Alan Jeffrey, and James Riely. 2020. Pomsets with preconditions: a simple model of relaxed memory. *Proc. ACM Program. Lang.* 4, Oopsla, Article 194 (Nov. 2020), 30 pages. https://doi.org/10.1145/3428262

Peter Jipsen and M. Andrew Moshier. 2016. Concurrent Kleene algebra with tests and branching automata. *Journal of Logical and Algebraic Methods in Programming* 85, 4 (2016), 637–652. https://doi.org/10.1016/j.jlamp.2015.12.005

Claire Jones. 1990. *Probabilistic Non-determinism*. Ph.D. Dissertation. University of Edinburgh. http://hdl.handle.net/1842/413

Claire Jones and Gordon Plotkin. 1989. A Probabilistic Powerdomain of Evaluations. In *Fourth Annual Symposium on Logic in Computer Science*. 186–195. https://doi.org/10.1109/lics.1989.39173

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018). https://doi.org/10.1017/S0956796818000151

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. Association for Computing Machinery, New York, NY, USA, 637–650. https://doi.org/10.1145/2676726.2676980

Benjamin Lucien Kaminski. 2019. *Advanced weakest precondition calculi for probabilistic programs*. Dissertation. RWTH Aachen University, Aachen. https://doi.org/10.18154/RWTH-2019-01829

Tobias Kappé, Paul Brunet, Alexandra Silva, Jana Wagemaker, and Fabio Zanasi. 2020. *Concurrent Kleene Algebra with Observations: From Hypotheses to Completeness*. Springer International Publishing, 381–400. https://doi.org/10.1007/978-3-030-45231-5_20

Joost-Pieter Katoen. 1996. *Quantitative and Qualitative Extensions of Event Structures*. PhD Thesis - Research UT, graduation UT. University of Twente, Netherlands. https://doi.org/10.3990/1.9789036507998

Ryan Kavanagh and Stephen Brookes. 2018. A Denotational Semantics for SPARC TSO. *Electronic Notes in Theoretical Computer Science* 336 (2018), 223–239. https://doi.org/10.1016/j.entcs.2018.03.025 The Thirty-third Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXIII).

Klaus Keimel and Gordon Plotkin. 2017. Mixed powerdomains for probability and nondeterminism. *Logical Methods in Computer Science* Volume 13, Issue 1 (Jan. 2017). https://doi.org/10.23638/LMCS-13(1:2)2017

Dexter Kozen. 1979. Semantics of probabilistic programs. In *20th Annual Symposium on Foundations of Computer Science (SFCS '79)*. 101–114. https://doi.org/10.1109/SFCS.1979.38

Dexter Kozen. 1983. A Probabilistic PDL. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing (STOC '83)*. Association for Computing Machinery, New York, NY, USA, 291–297. https://doi.org/10.1145/800061.808758

Dexter Kozen and Alexandra Silva. 2024. *Multisets and Distributions*. Springer Nature Switzerland, Cham, 168–187. https://doi.org/10.1007/978-3-031-61716-4_11

Dexter Kozen and Matvey Soloviev. 2018. Coalgebraic Tools for Randomness-Conserving Protocols. In *Relational*

and Algebraic Methods in Computer Science, Jules Desharnais, Walter Guttmann, and Stef Joosten (Eds.). Springer International Publishing, Cham, 298–313. https://doi.org/10.1016/j.jlamp.2021.100734

Andrei Lapets, Frederick Jansen, Kinan Dak Albab, Rawane Issa, Lucy Qin, Mayank Varia, and Azer Bestavros. 2018. Accessible Privacy-Preserving Web-Based Data Analysis for Assessing and Addressing Economic Inequalities. In *Proceedings of the 1st ACM SIGCAS Conference on Computing and Sustainable Societies* (Menlo Park and San Jose, CA, USA) *(COMPASS '18)*. Association for Computing Machinery, New York, NY, USA, Article 48, 5 pages. https://doi.org/10.1145/3209811.3212701

Michael R. Laurence and Georg Struth. 2014. Completeness Theorems for Bi-Kleene Algebras and Series-Parallel Rational Pomset Languages. In *Relational and Algebraic Methods in Computer Science*, Peter Höfner, Peter Jipsen, Wolfram Kahl, and Martin Eric Müller (Eds.). Springer International Publishing, Cham, 65–82. https://doi.org/10.1007/978-3-319-06251-8_5

Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. 2022. Finding Real Bugs in Big Programs with Incorrectness Logic. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 81 (Apr 2022), 27 pages. https://doi.org/10.1145/3527325

Dongjae Lee, Janggun Lee, Taeyoung Yoon, Minki Cho, Jeehoon Kang, and Chung-Kil Hur. 2025. Lilo: A Higher-Order, Relational Concurrent Separation Logic for Liveness. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 125 (April 2025), 28 pages. https://doi.org/10.1145/3720525

Daniel Lehmann and Michael O. Rabin. 1981. On the advantages of free choice: a symmetric and fully distributed solution to the dining philosophers problem. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Williamsburg, Virginia) *(POPL '81)*. Association for Computing Machinery, New York, NY, USA, 133–138. https://doi.org/10.1145/567532.567547

James Li. 2025. *Total Outcome Logic: Unified Reasoning For Nontermination and Branching Effects.* Master's thesis. Cornell University. https://doi.org/10.7298/sxsm-3458

James Li, Noam Zilberstein, and Alexandra Silva. 2025b. Total Outcome Logic: Unified Reasoning for a Taxonomy of Program Logics. arXiv:2411.00197 [cs.LO] https://arxiv.org/abs/2411.00197

John M. Li, Amal Ahmed, and Steven Holtzen. 2023. Lilac: A Modal Separation Logic for Conditional Probability. *Proc. ACM Program. Lang.* 7, PLDI, Article 112 (jun 2023), 24 pages. https://doi.org/10.1145/3591226

John M. Li, Jon Aytac, Philip Johnson-Freyd, Amal Ahmed, and Steven Holtzen. 2024. A Nominal Approach to Probabilistic Separation Logic. In *Proceedings of the 39th Annual ACM/IEEE Symposium on Logic in Computer Science* (Tallinn, Estonia) *(LICS '24)*. Association for Computing Machinery, New York, NY, USA, Article 55, 14 pages. https://doi.org/10.1145/3661814.3662135

Kwing Hei Li, Alejandro Aguirre, Simon Oddershede Gregersen, Philipp G. Haselwarter, Joseph Tassarotti, and Lars Birkedal. 2025a. Modular Reasoning about Error Bounds for Concurrent Probabilistic Programs. *Proc. ACM Program. Lang.* 9, ICFP, Article 245 (Aug. 2025), 30 pages. https://doi.org/10.1145/3747514

Hongjin Liang and Xinyu Feng. 2016. A program logic for concurrent objects under fair scheduling. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 385–399. https://doi.org/10.1145/2837614.2837635

Hongjin Liang and Xinyu Feng. 2017. Progress of concurrent objects with partial methods. *Proc. ACM Program. Lang.* 2, POPL, Article 20 (Dec. 2017), 31 pages. https://doi.org/10.1145/3158108

Yehuda Lindell. 2020. Secure multiparty computation. *Commun. ACM* 64, 1 (Dec. 2020), 86–96. https://doi.org/10.1145/3387108

Richard J. Lipton. 1977. A necessary and sufficient condition for the existence of hoare logics. In *18th Annual Sym-*

posium on Foundations of Computer Science (sfcs 1977). 1–6. https://doi.org/10.1109/SFCS.1977.1

Janine Lohse, Tim Rohde, Jimmy Xin, Niklas Mück, Iona Kuhn, Derek Dreyer, Deepak Garg, and Emanuele D'Osualdo. 2026. First Steps Towards Probabilistic Iris: Harmonizing Independence, Conditioning, and Dynamic Heap Allocation. arXiv:2605.13765 [cs.LO] https://arxiv.org/abs/2605.13765

Rupak Majumdar and V.R. Sathiyanarayana. 2025. Sound and Complete Proof Rules for Probabilistic Termination. *Proc. ACM Program. Lang.* 9, POPL, Article 63 (Jan. 2025), 32 pages. https://doi.org/10.1145/3704899

Zohar Manna and Amir Pnueli. 1974. Axiomatic Approach to Total Correctness of Programs. *Acta Inf.* 3, 3 (sep 1974), 243–263. https://doi.org/10.1007/BF00288637

George Markowsky. 1976. Chain-complete posets and directed sets with applications. *Algebra Universalis* 6 (12 1976), 53–68. https://doi.org/10.1007/BF02485815

The mathlib Community. 2020. The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs* (New Orleans, LA, USA) *(CPP 2020)*. Association for Computing Machinery, New York, NY, USA, 367–381. https://doi.org/10.1145/3372885.3373824

Annabelle McIver and Carroll Morgan. 2001. Partial correctness for probabilistic demonic programs. *Theoretical Computer Science* 266, 1 (2001), 513–541. https://doi.org/10.1016/S0304-3975(00)00208-5

Annabelle McIver and Carroll Morgan. 2005. *Abstraction, Refinement and Proof for Probabilistic Systems*. Springer. https://doi.org/10.1007/b138392

Annabelle McIver, Carroll Morgan, Benjamin Lucien Kaminski, and Joost-Pieter Katoen. 2018. A New Proof Rule for Almost-Sure Termination. *Proc. ACM Program. Lang.* 2, POPL, Article 33 (Jan 2018), 28 pages. https://doi.org/10.1145/3158121

Annabelle McIver, Tahiry Rabehaja, and Georg Struth. 2016. Probabilistic rely-guarantee calculus. *Theoretical Computer Science* 655 (2016), 120–134. https://doi.org/10.1016/j.tcs.2016.01.016

J.-J.Ch. Meyer and E.P. de Vink. 1988. Applications of compactness in the Smyth powerdomain of streams. *Theoretical Computer Science* 57, 2 (1988), 251–282. https://doi.org/10.1016/0304-3975(88)90042-4

J. J. Ch. Meyer and E. P. de Vink. 1989. Pomset semantics for true concurrency with synchronization and recursion. In *Mathematical Foundations of Computer Science 1989*, Antoni Kreczmar and Grazyna Mirkowska (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 360–369. https://doi.org/10.1007/3-540-51486-4_83

Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. 2016. The Honey Badger of BFT Protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 31–42. https://doi.org/10.1145/2976749.2978399

E. Moggi. 1989. Computational lambda-calculus and monads. In *[1989] Proceedings. Fourth Annual Symposium on Logic in Computer Science*. 14–23. https://doi.org/10.1109/LICS.1989.39155

Eugenio Moggi. 1991. Notions of computation and monads. *Information and Computation* 93, 1 (1991), 55–92. https://doi.org/10.1016/0890-5401(91)90052-4

Alexandre Moine, Sam Westrick, and Joseph Tassarotti. 2026. All for One and One for All: Program Logics for Exploiting Internal Determinism in Parallel Programs. *Proc. ACM Program. Lang.* 10, POPL, Article 26 (Jan. 2026), 29 pages. https://doi.org/10.1145/3776668

Carroll Morgan, Annabelle McIver, and Karen Seidel. 1996a. Probabilistic Predicate Transformers. *ACM Trans. Program. Lang. Syst.* 18, 3 (may 1996), 325–353. https://doi.org/10.1145/229542.229547

Carroll Morgan, Annabelle McIver, Karen Seidel, and J. W. Sanders. 1996b. Refinement-oriented probability for CSP.

Form. Asp. Comput. 8, 6 (nov 1996), 617–647. https://doi.org/10.1007/bf01213492

Robert Morris. 1978. Counting large numbers of events in small registers. *Commun. ACM* 21, 10 (Oct. 1978), 840–842. https://doi.org/10.1145/359619.359627

Rajeev Motwani and Prabhakar Raghavan. 1995. *Randomized Algorithms*. Cambridge University Press.

Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings*. Springer-Verlag, Berlin, Heidelberg, 625–635. https://doi.org/10.1007/978-3-030-79876-5_37

Renato Neves. 2025. An Adequacy Theorem Between Mixed Powerdomains and Probabilistic Concurrency. *Electronic Proceedings in Theoretical Computer Science* 433 (Oct 2025), 23–45. https://doi.org/10.4204/eptcs.433.4

Peter W. O'Hearn. 2004. Resources, Concurrency and Local Reasoning. In *CONCUR 2004 - Concurrency Theory*. Springer Berlin Heidelberg, Berlin, Heidelberg, 49–67. https://doi.org/10.1016/j.tcs.2006.12.035

Peter W. O'Hearn. 2020. Incorrectness Logic. *Proc. ACM Program. Lang.* 4, POPL, Article 10 (Jan. 2020), 32 pages. https://doi.org/10.1145/3371078

Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs That Alter Data Structures. In *Proceedings of the 15th International Workshop on Computer Science Logic (CSL '01)*. Springer-Verlag, Berlin, Heidelberg, 1–19. https://doi.org/10.1007/3-540-44802-0_1

Louis Parlant. 2020. *Monad Composition via Preservation of Algebras*. Ph.D. Dissertation. University College London. https://discovery.ucl.ac.uk/id/eprint/10112228/

Gordon Plotkin. 1976. A Powerdomain Construction. *SIAM J. Comput.* 5, 3 (1976), 452–487. https://doi.org/10.1137/0205035

Vaughan R. Pratt. 1976. Semantical Considerations on Floyd-Hoare Logic. In *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*. 109–121. https://doi.org/10.1109/SFCS.1976.27

William Pugh. 1990. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM* 33, 6 (June 1990), 668–676. https://doi.org/10.1145/78973.78977

Azalea Raad, Josh Berdine, Hoang-Hai Dang, Derek Dreyer, Peter O'Hearn, and Jules Villard. 2020. Local Reasoning About the Presence of Bugs: Incorrectness Separation Logic. In *Computer Aided Verification*. Springer International Publishing, Cham, 225–252. https://doi.org/10.1007/978-3-030-53291-8_14

Robert Rand and Steve Zdancewic. 2015. VPHL: A Verified Partial-Correctness Logic for Probabilistic Programs. In *Electronic Notes in Theoretical Computer Science*, Vol. 319. 351–367. https://doi.org/10.1016/j.entcs.2015.12.021

C. Radhakrishna Rao. 1961. Generation of Random Permutations of Given Number of Elements Using Random Sampling Numbers. *Sankhyā: The Indian Journal of Statistics, Series A (1961-2002)* 23, 3 (1961), 305–307. http://www.jstor.org/stable/25049166

J.C. Reynolds. 2002. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. 55–74. https://doi.org/10.1109/LICS.2002.1029817

H. L. Royden. 1968. *Real Analysis* (2d ed. ed.). Macmillan, New York.

Martin Sandelius. 1962. A Simple Randomization Procedure. *Journal of the Royal Statistical Society: Series B (Methodological)* 24, 2 (12 1962), 472–481. https://doi.org/10.1111/j.2517-6161.1962.tb00474.x

Dana Scott. 1972. Continuous lattices. In *Toposes, Algebraic Geometry and Logic*, F. W. Lawvere (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 97–136. https://doi.org/10.1007/BFb0073967

Dana Scott and Christopher Strachey. 1971. *Toward A Mathematical Semantics For Computer Languages.* Technical Report Prg06. OUCL. 49 pages.

Adi Shamir. 1979. How to share a secret. *Commun. ACM* 22, 11 (Nov. 1979), 612–613. https://doi.org/10.1145/359168.359176

Victor Shoup. 2000. Practical Threshold Signatures. In *Advances in Cryptology — EUROCRYPT 2000*, Bart Preneel (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 207–220. https://doi.org/10.1007/3-540-45539-6_15

Michael Smyth. 1978. Power domains. *J. Comput. System Sci.* 16, 1 (1978), 23–36. https://doi.org/10.1016/0022-0000(78)90048-X

Harald Søndergaard and Peter Sestoft. 1992. Non-determinism in Functional Languages. *Comput. J.* 35, 5 (10 1992), 514–523. https://doi.org/10.1093/comjnl/35.5.514

Joseph Tassarotti. 2018. *Verifying Concurrent Randomized Algorithms.* Ph.D. Dissertation. Carnegie Mellon University. https://csd.cmu.edu/academics/doctoral/degrees-conferred/joseph-tassarotti

Joseph Tassarotti and Robert Harper. 2019. A Separation Logic for Concurrent Randomized Programs. *Proc. ACM Program. Lang.* 3, POPL, Article 64 (Jan 2019), 30 pages. https://doi.org/10.1145/3290377

Regina Tix. 1999. *Continuous D-cones: convexity and powerdomain constructions.* Ph.D. Dissertation. Darmstadt University of Technology, Germany. https://d-nb.info/957239157

Regina Tix. 2000. Convex Power Constructions for Continuous D-Cones. *Electronic Notes in Theoretical Computer Science* 35 (2000), 206–229. https://doi.org/10.1016/S1571-0661(05)80746-7

Regina Tix, Klaus Keimel, and Gordon Plotkin. 2009. Semantic Domains for Combining Probability and Non-Determinism. *Electronic Notes in Theoretical Computer Science* 222 (2009), 3–99. https://doi.org/10.1016/j.entcs.2009.01.002

Viktor Vafeiadis. 2011. Concurrent Separation Logic and Operational Semantics. *Electronic Notes in Theoretical Computer Science* 276 (2011), 335–351. https://doi.org/10.1016/j.entcs.2011.09.029

Daniele Varacca. 2002. The powerdomain of indexed valuations. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science.* 299–308. https://doi.org/10.1109/LICS.2002.1029838

Daniele Varacca. 2003. *Probability, Nondeterminism and Concurrency: Two Denotational Models for Probabilistic Computation.* Ph.D. Dissertation. University of Aarhus. https://www.brics.dk/DS/03/14/

Daniele Varacca, Hagen Völzer, and Glynn Winskel. 2006. Probabilistic event structures and domains. *Theoretical Computer Science* 358, 2 (2006), 173–199. https://doi.org/10.1016/j.tcs.2006.01.015

Daniele Varacca and Glynn Winskel. 2006. Distributing probability over non-determinism. *Mathematical Structures in Computer Science* 16, 1 (2006), 87–113. https://doi.org/10.1017/S0960129505005074

Daniele Varacca and Nobuko Yoshida. 2007. Probabilistic π-Calculus and Event Structures. *Electronic Notes in Theoretical Computer Science* 190, 3 (2007), 147–166. https://doi.org/10.1016/j.entcs.2007.07.009

John von Neumann. 1951. Various techniques used in connection with random digits. In *Monte Carlo Method*, A.S. Householder, G.E. Forsythe, and H.H. Germond (Eds.). National Bureau of Standards Applied Mathematics Series, 12, Washington, D.C.: U.S. Government Printing Office, 36–38.

Philip Wadler. 1993. Monads for functional programming. In *Program Design Calculi.* Springer Berlin Heidelberg, Berlin, Heidelberg, 233–264. https://doi.org/10.1007/978-3-662-02880-3_8

Glynn Winskel. 2014. *Probabilistic and Quantum Event Structures.* Springer International Publishing, Cham, 476–497. https://doi.org/10.1007/978-3-319-06880-0_25

Hongseok Yang and Peter O'Hearn. 2002. A Semantic Basis for Local Reasoning. In *Foundations of Software Science and Computation Structures*. Springer Berlin Heidelberg, Berlin, Heidelberg, 402–416. https://doi.org/10.1007/3-540-45931-6_28

Andrew C. Yao. 1982. Protocols for Secure Computations. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society, Los Alamitos, CA, USA, 160–164. https://doi.org/10.1109/SFCS.1982.88

Juan Yu and Costas Zarifis. 2021. Use HyperLogLog for trend analysis with Amazon Redshift. https://aws.amazon.com/blogs/big-data/use-hyperloglog-for-trend-analysis-with-amazon-redshift/ AWS Big Data Blog, accessed 2026.

Linpeng Zhang, Noam Zilberstein, Benjamin Lucien Kaminski, and Alexandra Silva. 2024. Quantitative Weakest Hyper Pre: Unifying Correctness and Incorrectness Hyperproperties via Predicate Transformers. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 300 (oct 2024), 30 pages. https://doi.org/10.1145/3689740

Noam Zilberstein. 2020. Eliminating Bugs with Dependent Haskell (Experience Report). In *Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell* (Virtual Event, USA) *(Haskell 2020)*. Association for Computing Machinery, New York, NY, USA, 9. https://doi.org/10.1145/3406088.3409020

Noam Zilberstein. 2024. Unified Analysis Techniques for Programs with Outcomes. In *Companion Proceedings of the 2024 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity* (Pasadena, CA, USA) *(SPLASH Companion '24)*. Association for Computing Machinery, New York, NY, USA, 4–6. https://doi.org/10.1145/3689491.3691814

Noam Zilberstein. 2025. Outcome Logic: A Unified Approach to the Metatheory of Program Logics with Branching Effects. *ACM Trans. Program. Lang. Syst.* 47, 3, Article 14 (Sept. 2025), 71 pages. https://doi.org/10.1145/3743131

Noam Zilberstein, Derek Dreyer, and Alexandra Silva. 2023. Outcome Logic: A Unifying Foundation for Correctness and Incorrectness Reasoning. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 93 (Apr 2023), 29 pages. https://doi.org/10.1145/3586045

Noam Zilberstein, Daniele Gorla, and Alexandra Silva. 2025a. Denotational Semantics for Probabilistic and Concurrent Programs. In *36th International Conference on Concurrency Theory (CONCUR 2025) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 348)*, Patricia Bouyer and Jaco van de Pol (Eds.). Schloss Dagstuhl – Leibniz-Zentrum f"ur Informatik, Dagstuhl, Germany, 39:1–39:24. https://doi.org/10.4230/LIPIcs.CONCUR.2025.39

Noam Zilberstein, Dexter Kozen, Alexandra Silva, and Joseph Tassarotti. 2025b. A Demonic Outcome Logic for Randomized Nondeterminism. *Proc. ACM Program. Lang.* 9, POPL, Article 19 (Jan 2025), 30 pages. https://doi.org/10.1145/3704855

Noam Zilberstein, Angelina Saliling, and Alexandra Silva. 2024. Outcome Separation Logic: Local Reasoning for Correctness and Incorrectness with Computational Effects. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 104 (apr 2024), 29 pages. https://doi.org/10.1145/3649821

Noam Zilberstein, Alexandra Silva, and Joseph Tassarotti. 2026. Probabilistic Concurrent Reasoning in Outcome Logic: Independence, Conditioning, and Invariants. *Proc. ACM Program. Lang.* 10, POPL (Jan 2026), 30 pages. https://doi.org/10.1145/3776651

Maaike Zwart. 2020. *On the Non-Compositionality of Monads via Distributive Laws*. Ph.D. Dissertation. University of Oxford. https://ora.ox.ac.uk/objects/uuid:b2222b14-3895-4c87-91f4-13a8d046febb

Maaike Zwart and Dan Marsden. 2019. No-Go Theorems for Distributive Laws. In *2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. 1–13. https://doi.org/10.1109/lics.2019.8785707