Denotational Semantics for Probabilistic and Concurrent Programs

Noam Zilberstein
Cornell University, Ithaca, NY, USA

Daniele Gorla
Sapienza Università di Roma, Italy

Alexandra Silva
Cornell University, Ithaca, NY, USA
University College London, UK

Abstract

We develop a denotational model for *probabilistic and concurrent imperative programs*, a class of programs with standard control flow via conditionals and while-loops, as well as probabilistic actions and parallel composition. Whereas semantics for concurrent or randomized programs in isolation is well studied, their combination has not been thoroughly explored and presents unique challenges. The crux of the problem is that interactions between control flow, probabilistic actions, and concurrent execution cannot be captured by straightforward generalizations of prior work on pomsets and convex languages, prominent models for those effects, individually. Our model has good domain theoretic properties, important for semantics of unbounded loops. We also prove two adequacy theorems, showing that the model subsumes typical powerdomain semantics for concurrency and convex powerdomain semantics for probabilistic nondeterminism.

2012 ACM Subject Classification Theory of computation → Parallel computing models; Theory of computation → Denotational semantics; Theory of computation → Probabilistic computation

Keywords and phrases Denotational Semantics, Pomsets, Concurrency, Convex Powerset

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2025.39

Related Version *Full Version*: `https://doi.org/10.48550/arXiv.2503.02768` [67]

Funding This work was partially supported ARIA's Safeguarded AI programme.

1 Introduction

From simple imperative languages, to functional ones, to probabilistic and quantum paradigms, the development of programming languages is often accompanied by a study of mathematical objects capturing the semantics of syntactic constructs. Program semantics underpins many static analysis and verification techniques, and also how a language can be extended.

Semantics can be presented in different styles – denotational, operational, and axiomatic – and each approach offers different insights. Denotational semantics is often chosen for expressivity and extensibility. For example, adding recursion raises questions about the underlying *domain*, motivating the development of domain theory to provide mathematical representations of iterated computations [51, 52]. Other constructs necessitate more involved domains; two prime examples are probabilistic and concurrent programs.

Probabilistic programs have been widely studied – going back to the seminal works of Kozen [31] and McIver and Morgan [36] – and their semantic domains require extra algebraic structure to capture the *distribution* of outcomes generated by operations such as sampling and coin flips. This added convex structure brings additional complexity to questions of language extensions and program analysis, but the last decade has brought success stories on taming the complexity through program logics and predicate transformer calculi [5, 10, 36, 41, 68].

Concurrent programs have prominent applications in distributed systems and have long been a target of program analysis, as concurrency bugs are hard to detect and prevent. There have been many approaches to concurrent semantics, some arising from process algebra research and some from more practical hardware considerations (*e.g.,* memory models). In terms of denotational semantics, *pomsets* [8,17,18,47] provide a prominent model, expressing causality between actions and parallel branching as a partial order.

Semantics of concurrent and probabilistic programs, separately, are subtle and complex. Unsurprisingly, their combination introduces additional challenges. In this paper, we develop a denotational model for programs that mix probabilistic and concurrent constructs. We work with a simple concurrent imperative programming language, shown below.

$$\mathit{cmd} \ni C ::= \textsf{skip} \mid C_1; C_2 \mid \; C_1 \,|\, C_2 \; \mid \textsf{if } b \textsf{ then } C_1 \textsf{ else } C_2 \mid \textsf{while } b \textsf{ do } C \mid a \qquad (1)$$

Here, the basic actions $a \in \mathit{act}$ can perform probabilistic operations such as random sampling. This is not only interesting from a theoretical point of view; for decades, randomization has been used to enhance the capabilities of concurrent programs [19,43,49,50]. For example, Dijkstra's famous *Dining Philosophers Problem* – a distributed synchronization scheme – has no deterministic solution [11], but it has a simple randomized one [35].

Despite the importance of mixing randomization and concurrency, semantic models for probabilistic concurrent programs with unbounded looping are not yet well understood. Such models are necessary to analyze behaviors of many concurrent randomized protocols – including the Dining Philosophers – which avoid deadlock with probability 1, but admit an infinite trace whose probability converges to 0. In this paper, we develop such a model, drawing insights from prior work on concurrent and probabilistic semantics, but solving challenges unique to their combination. In a nutshell, the contributions of this paper are:

1. We introduce *pomsets with formulae*, and prove that they have well behaved domain-theoretic properties (Section 3), including an *extension lemma* [37], for extending monotone operations on finite pomsets to continuous operations on infinite ones.
2. We define guarded, sequential, and parallel composition operations on pomsets with formulae (Section 4) and use those operations to define the denotational semantics for the concurrent imperative language (1), with uninterpreted actions (Section 5.1).
3. We define a linearization operation (via the extension lemma), parametric on the computational domain, to convert the pomset semantics into a state transformer (Section 5.2).
4. We prove two adequacy theorems showing that our model captures established models for probabilistic nondeterminism (Theorem 5.3) and pure concurrency (Theorem 6.1).

We discuss related work in Section 7; omitted proofs and details are provided in [67].

2 An Overview of Probabilistic Concurrency Semantics

In this section, we introduce the requirements for interpreting programs that are both probabilistic and concurrent. To illustrate the subtleties, consider the following program:

$$C \triangleq x :\approx \textsf{flip}\left(\tfrac{1}{2}\right); (y := 0 \,|\, y := 1); \textsf{if } y \textsf{ then skip else } x := x + 2 \qquad (2)$$

After executing C, the value of x can be any integer between 0 and 3, and that value depends both on random sampling and nondeterminism (from concurrent scheduling). The parity of x is fixed after the coin flip (returning 0 or 1 with probability $\frac{1}{2}$), but then the scheduler can influence its final value by choosing in which order to run the updates to y. Importantly though, x will be even with probability exactly $\frac{1}{2}$, regardless of the behavior of the scheduler.

Any semantic domain for this program must encompass probability distributions of program states, while at the same time accounting for nondeterminism introduced from parallel branching. One might attempt to give semantics to this program using a naive composition of distributions and powerset (as monads) but, as it is well known from prior work, such combination can easily lead to non-compositional semantics [45, 63, 71, 72]. While there are several domains for this combination of effects, we use the *convex powerset*, which is quite well-studied in that it is a monad [23] for sequential composition; it is a directed complete partial order (DCPO) [30, 57–59] for finding fixed points of iterated computations; and has well behaved equational laws [6, 39, 40].

We now give a basic account of the convex powerset; for more details on the DCPO structure and monad operations, refer to Section A.2 and [68, §3]. A discrete probability distribution $\mu \in \mathcal{D}(X) = X \to [0,1]$ is a countable map from X to probabilities such that $\sum_{x\in X} \mu(x) = 1$. Convex combinations are defined for distributions $(\mu \oplus_p \nu)(x) = p \cdot \mu(x) + (1-p) \cdot \nu(x)$ and sets of distributions $S \oplus_p T \triangleq \{\mu \oplus_p \nu \mid \mu \in S, \nu \in T\}$. A set $S \subseteq \mathcal{D}(X)$ is *convex* if $S = S \oplus_p S$ for all $p \in [0,1]$, *i.e.,* S contains all convex combinations of its elements. Our computational domain – *the convex powerset* – consists of nonempty convex sets of distributions[1].

$$\mathcal{C}(X) \triangleq \{S \subseteq \mathcal{D}(X \cup \{\bot\}) \mid S \text{ is nonempty, convex, } \ldots\}$$

This computational domain allows us give semantics to the coin flip actions in program (2) $[\![a]\!]_{\mathit{act}} : \mathcal{S} \to \mathcal{C}(\mathcal{S})$ where $\mathcal{S} \triangleq \mathsf{var} \to \mathsf{val}$ is the set of variable valuations. The semantics is given by a set containing a single distribution, where x is set to 1 with probability p and it is set to 0 with probability $1-p$. Being a *set of distributions*, $\mathcal{C}(\mathcal{S})$ can also represent nondeterministic choice, which we denote by &. Operationally, $S \mathbin{\&} T$ is not simply a choice between S and T, but rather it is a choice of a *distribution* over S and T, corresponding to a scheduler that can use biased coin flips to make decisions [60].

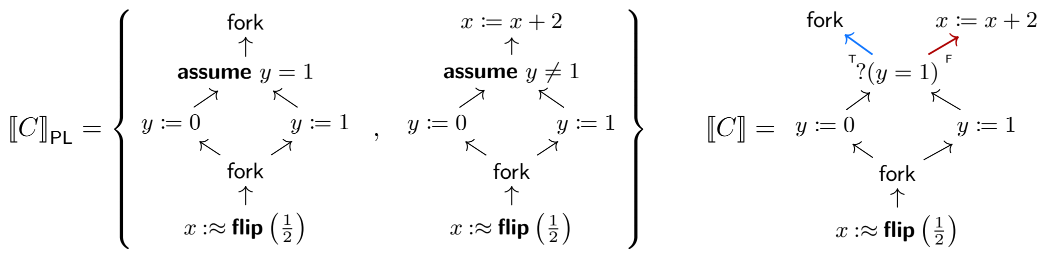
$$[\![x :\approx \textsf{flip}\,(p)]\!]_{\mathit{act}}\,(s) \triangleq \left\{ \begin{bmatrix} s[x := 1] & \mapsto & p \\ s[x := 0] & \mapsto & 1-p \end{bmatrix} \right\} \qquad\qquad S \mathbin{\&} T \triangleq \bigcup_{p\in[0,1]} S \oplus_p T$$

Our goal is to develop a pomset model to compositionally reason about concurrency in a manner that is compatible with the convex powerset. Existing pomset models do not track control flow, and instead rely on the following equation, stating that a command followed by an if-statement can be decomposed into two traces, with the split lifted to the top:

$$C; \textsf{if}\ b\ \textsf{then}\ C_1\ \textsf{else}\ C_2 \quad\equiv\quad (C; \textsf{assume}\ b; C_1) \mathbin{\&} (C; \textsf{assume}\ \neg b; C_2) \tag{3}$$

It is well known that this equation is invalid in probabilistic contexts, especially $\mathcal{C}$ [39]. More concretely, prior denotational models for concurrency $[\![-]\!]_{\mathsf{PL}} : \mathit{cmd} \to \mathcal{P}(\mathit{pom})$ (shown in Figure 5) map programs to *pomset languages* – sets of pomsets – where each $\boldsymbol{\alpha} \in [\![C]\!]_{\mathsf{PL}}$ corresponds to a particular resolution of the tests in the program. For program (2), this approach yields the semantics on the left of Figure 1, where the problematic Equation (3) is implicitly applied. Arrows $a_1 \to a_2$ indicate causality, *i.e.,* a_1 must occur before a_2; when there is no arrow between two actions, they can be interleaved in any order. There is a significant problem in $[\![C]\!]_{\mathsf{PL}}$; as we will soon see, there is no straightforward way to piece together the entire distribution of outcomes in the program.

[1] The DCPO structure relies of a few more properties, which we report in Section A.2.



$[\![C]\!]_{\mathsf{PL}} =$
fork
assume $y = 1$
$y := 0$
$y := 1$
fork
$x :\approx$ flip $\left(\frac{1}{2}\right)$
$x := x + 2$
assume $y \neq 1$
$y := 0$
$y := 1$
fork
$x :\approx$ flip $\left(\frac{1}{2}\right)$
$[\![C]\!] =$
fork
$x := x + 2$
T
$?(y = 1)$
F
$y := 0$
$y := 1$
fork
$x :\approx$ flip $\left(\frac{1}{2}\right)$

Figure 1 Pomset language model (left) vs pomset with formulae model (right) for program (2) (we use a fork with 0 branches to represent **skip**).

Crucially, our pomset structure tracks the results of control-flow tests (which can rely on randomization). This is done symbolically by associating a Boolean formula to each node of the pomset, recording the resolution of tests needed to reach that point. We call these new structures *pomsets with formulae*, which capture both parallel composition and guarded branching[2]. As such, this structure encodes many traces, avoiding the need for a set of traces and simultaneously remaining compositional in the presence of probabilistic actions.

The semantics in our new model $[\![-]\!] : \mathit{cmd} \to \mathit{pom}$ for program (2) is shown on the right of Figure 1. Compared to $[\![C]\!]_{\mathsf{PL}}$, our new model has merged the set of structures into a single structure, where the two opposing tests are replaced by a single node labelled $?(y = 1)$. The outgoing edges are labelled T and F, indicating that they can only be followed if the test passes or fails, respectively. Now, we may wish to know the possible probabilities for each outcome resulting from running this program. To this end, in Section 5.2 we develop a *linearization* procedure $\mathcal{L} \colon \mathit{pom} \to \mathcal{S} \to \mathcal{C}(\mathcal{S})$ for interpreting a pomset as a state transformer. Linearizing this structure, we get the following set of distributions:

$$\mathcal{L}([\![C]\!])(s) = \left\{ \left[\begin{array}{ll} s[x := 0, y := 0] & \mapsto p \\ s[x := 1, y := 0] & \mapsto q \\ s[x := 2, y := 1] & \mapsto \frac{1}{2} - p \\ s[x := 3, y := 1] & \mapsto \frac{1}{2} - q \end{array} \right] \;\middle|\; 0 \leq p, q \leq \frac{1}{2} \right\}$$

Linearization provides new insights about the program, which were not obvious in the pomset structure. The scheduler can make y equal to 0 or 1 with any probability, which matches our operational understanding of the program: the scheduler can choose to execute the commands in either the order $y := 0; y := 1$ or $y := 1; y := 0$ (or some convex combination thereof). However, regardless of the scheduler's choice of p and q, x is even (or odd) with exactly probability $\frac{1}{2}$. This formal semantics matches the intuition for the program, since the parity of x is fixed by the initial coin flip.

We now contrast the semantics above with pomset language semantics to show that the program C cannot be meaningfully interpreted using prior techniques. Recall that in $[\![C]\!]_{\mathsf{PL}}$ each pomset corresponds to a particular resolution of the test, *i.e.*, $y = 1$ and $y \neq 1$. So, in the first pomset we will always have $x \leq 1$ whereas in the second pomset $x \geq 2$. Let us now try to obtain a meaningful state transformer semantics from this program by linearizing each structure and then merging the results. To do so, we use the following semantics for **assume**,

[2] Formulae are more expressive than conflict relations, *e.g.*, in event structures: conflict can only exclude execution of branches, whereas formulae record the entire outcome of tests that lead to every node.

where failed tests evaluate to a special **?** symbol, which halts the program execution:

$$[\![\textsf{assume}\ b]\!]_{\textit{act}}(s) = \begin{cases} \eta(s) & \text{if } [\![b]\!]_{\textit{test}}(s) = 1 \\ \eta(\textbf{?}) & \text{if } [\![b]\!]_{\textit{test}}(s) = 0 \end{cases}$$

Now, letting $\boldsymbol{\alpha}_1$ and $\boldsymbol{\alpha}_2$ be the left and right pomsets in $[\![C]\!]_{\mathsf{PL}}$, we apply linearization to both structures to obtain:

$$\mathcal{L}(\boldsymbol{\alpha}_1)(s) = \left\{ \left[\begin{array}{ccc} s[x := 0, y := 0] & \mapsto & p_1 \\ s[x := 1, y := 0] & \mapsto & q_1 \\ \textbf{?} & \mapsto & 1 - p_1 - q_1 \end{array} \right] \middle| \ 0 \le p_1, q_1 \le \frac{1}{2} \right\}$$

$$\mathcal{L}(\boldsymbol{\alpha}_2)(s) = \left\{ \left[\begin{array}{ccc} s[x := 2, y := 1] & \mapsto & p_2 \\ s[x := 3, y := 1] & \mapsto & q_2 \\ \textbf{?} & \mapsto & 1 - p_2 - q_2 \end{array} \right] \middle| \ 0 \le p_2, q_2 \le \frac{1}{2} \right\}$$

Already, we can begin to see a problem; in $\mathcal{L}([\![C]\!])(s)$ the scheduler picks two probabilities, but here the scheduler picks four probabilities, giving it a higher degree of freedom to influence the outcome of the program. We make this formal by attempting to define an operation $\bowtie$ to merge the two semantics. As a first attempt, we generate a new set of distributions by summing the non-**?** probability mass in every pair of distributions from the two sets such that the mass of **?** in one is equal to the non-**?** mass in the other:

$$S \bowtie T \triangleq \{\mu \bowtie \nu \mid \mu \in S, \nu \in T, \mu(\textbf{?}) = 1 - \nu(\textbf{?})\} \qquad (\mu \bowtie \nu)(s) \triangleq \begin{cases} \mu(s) + \nu(s) & \text{if } s \neq \textbf{?} \\ 0 & \text{if } s = \textbf{?} \end{cases}$$

Indeed, this gives us $\mathcal{L}([\![C]\!])(s) \subseteq \mathcal{L}(\boldsymbol{\alpha}_1)(s) \bowtie \mathcal{L}(\boldsymbol{\alpha}_2)(s)$: take any $\mu \in \mathcal{L}([\![C]\!])(s)$, which is generated by fixing some probabilities $0 \le p, q \le \frac{1}{2}$. Picking $p_1 = p$, $q_1 = q$, $p_2 = \frac{1}{2} - p$, and $q_2 = \frac{1}{2} - q$, we see that $\mu \in \mathcal{L}(\boldsymbol{\alpha}_1)(s) \bowtie \mathcal{L}(\boldsymbol{\alpha}_2)(s)$. However, $\mathcal{L}(\boldsymbol{\alpha}_1)(s) \bowtie \mathcal{L}(\boldsymbol{\alpha}_2)(s)$ contains extra distributions that are not in $\mathcal{L}([\![C]\!])(s)$, and are not correct outcomes of the program. For example, letting $p_1 = p_2 = \frac{1}{2}$ and $q_1 = q_2 = 0$, we get:

$$\left[\begin{array}{ll} s[x := 0, y := 0] & \mapsto \frac{1}{2} \\ s[x := 1, y := 0] & \mapsto 0 \\ s[x := 2, y := 1] & \mapsto \frac{1}{2} \\ s[x := 3, y := 1] & \mapsto 0 \end{array} \right] \in \mathcal{L}(\boldsymbol{\alpha}_1)(s) \bowtie \mathcal{L}(\boldsymbol{\alpha}_2)(s)$$

But this distribution cannot be an outcome of running the program, since x is even with probability 1, deviating from the expected operational behavior!

So a different implementation of $\bowtie$ is needed, which is more discerning about the compatibility of pairs of distributions from the two sets. But as it stands, there is no information in $\mathcal{L}(\boldsymbol{\alpha}_1)(s)$ and $\mathcal{L}(\boldsymbol{\alpha}_2)(s)$ to indicate compatibility. While we do not claim that it is impossible to record this information during linearization, our investigation suggests that it would not be straightforward. A proper linearization of the pomset language $[\![C]\!]_{\mathsf{PL}}$ would essentially amount to a lockstep execution of the two pomsets, so that both branches can be taken with the appropriate probabilities. That lockstep execution corresponds to a straightforward linearization of the single pomset with formulae $[\![C]\!]$, where each test already has information about both branches. We therefore conclude that pomsets with formulae are the correct semantic structure for this domain. In the remainder of the paper, we develop the formal details of our pomset with formulae model and associated linearization procedure.

3 Semantic Structures

In this section, we define a particular kind of labelled partial order, which will form the core of our semantic model. We call this structure a *labeled partial order with formulae* (LPOF), as it includes a special labelling, assigning a Boolean formula to each node. We recall basic definitions from domain theory in Section A.1; for a complete treatment refer to [1].

3.1 Labelled Partial Orders with Formulae

Let *nodes* be a countable universe of nodes, that will be denoted by $x, y, z, \ldots$, whereas subsets of *nodes* will be denoted by $N, X, Y, \ldots$. We start by defining the class of Boolean formulae using nodes as the variables:

$$\mathit{form} \ni \psi ::= \mathsf{true} \mid \mathsf{false} \mid \psi_1 \wedge \psi_2 \mid \psi_1 \vee \psi_2 \mid \neg\psi \mid x$$

Formulae are interpreted by valuations $v \colon \mathit{nodes} \to \mathbb{B}$, where $\mathbb{B} = \{0, 1\}$; the satisfaction relation, written $v \vDash \psi$, is defined in the standard way (see Definition A.5 in Section A.1). 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 $\psi' \Rightarrow \psi$. In what follows, given a strict poset $\langle X, < \rangle$ and any $x \in X$, we let $x{\downarrow} \triangleq \{y \in X \mid y < x\}$, $x{\uparrow} \triangleq \{y \in X \mid x < y\}$, $\mathsf{succ}(x) \triangleq \{y \in X \mid x < y \wedge \nexists z.(x < z < y)\}$, and $\min \triangleq \{x \in X \mid x{\downarrow} = \emptyset\}$. Further, the level of x, written $\mathsf{lev}(x)$, is the length of the longest path from a minimal element to x. See Section A.1 for more details.

▶ **Definition 3.1** (LPOF). *Let $\langle L, \leq \rangle$ be a pointed, finitely preceded DCPO with bottom element $\bot$. A labelled partial order with formulae (LPOF) over L is a 4-tuple $\alpha = \langle N, <, \lambda, \varphi \rangle$ 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 $|x{\downarrow}| < \infty$, 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: $|\min| = 1$.*
3. *$\lambda \colon N \to L$ is a labelling function such that $\mathsf{succ}(x) = \emptyset$ whenever $\lambda(x) = \bot$.*
4. *$\varphi \colon N \to \mathit{form}$ is a formula function satisfying:*
   a. *$\mathsf{sat}(\varphi(x))$ and $\mathsf{vars}(\varphi(x)) \subseteq x{\downarrow}$, for all $x \in N$;*
   b. *$\varphi(y) \Rightarrow \varphi(x)$, for all $x < y$.*

We denote by $\mathit{lpof}(L)$ the set of all LPOFs over L.

For some LPOF $\alpha = \langle N, <, \lambda, \varphi \rangle$, we use N_α, $<_\alpha$, λ_α, and φ_α to refer to its constituent parts; similarly, we annotate all functions with the LPOF they refer to, *e.g.,* $x{\downarrow}_\alpha$, $\mathsf{succ}_\alpha(x)$, $\min_\alpha$, etc. We now explain the conditions in Definition 3.1.

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. With this in mind, conditions (2a) and (2b) are standard; in particular, since a node represents an action of a program, having finitely many predecessors ensures that every action may happen 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: indeed, we allow a node to have infinitely many successors, but they cannot be all at the same level. We defer the discussion on Condition (2c) to Remark 4.1 later on.

Concerning Condition (3), labels correspond to *Boolean tests* and *actions*, performed during a program execution (*e.g.,* Figure 1). Unlike standard pomset models, actions can be probabilistic, and our model is consistent with known sequential probabilistic semantics

(Theorem 5.3). For the moment, we leave labels unspecified, only assuming the presence of $\bot$, used to denote a nonterminating computation, which is fundamental in approximating the fixed point semantics of while-loops. Since $\bot$ denotes nontermination, nodes labelled with $\bot$ cannot be a predecessor of any node, as the successors of $\bot$ would never be executed.

Condition (4) is about formulae, which are crucial for modeling conditional constructs (if-then-else and while-loops), such as **if** b_1 **then** a_1 **else if** b_2 **then** a_2 **else** a_3, where b_1 and b_2 are tests and a_1, a_2, a_3 are actions. This program corresponds to the LPOF below, where $\lambda(x_\ell) = \ell$ for all $\ell \in \{b_1, b_2, a_1, a_2, a_3\}$, $w \xrightarrow{\mathsf{T}} w_1$ means that $\varphi(w_1) \Leftrightarrow \varphi(w) \wedge w$ and $w \xrightarrow{\mathsf{F}} w_2$ means that $\varphi(w_2) \Leftrightarrow \varphi(w) \wedge \neg w$ (*i.e.,* w is labeled with a test, whose outcome leads to w_1 or w_2, depending on the Boolean outcome depicted on the colored arc):

$$\begin{tikzcd}
\varphi(x_{a_2}) = \neg x_{b_1} \wedge x_{b_2} & & x_{a_2} & & x_{a_3} & \varphi(x_{a_3}) = \neg x_{b_1} \wedge \neg x_{b_2} \\
\varphi(x_{a_1}) = x_{b_1} & x_{a_1} & & x_{b_2} \arrow[ul, "\mathsf{T}"'] \arrow[ur, "\mathsf{F}"] & & \varphi(x_{b_2}) = \neg x_{b_1} \\
\varphi(x_{b_1}) = \mathsf{true} & & x_{b_1} \arrow[ul, "\mathsf{T}"'] \arrow[ur, "\mathsf{F}"] & & &
\end{tikzcd} \tag{4}$$

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. With this in mind, Definition 3.1(4) is quite intuitive:

- Requiring $\mathsf{sat}(\varphi(x))$ amounts to expressing the fact that every node x is reachable, *i.e.,* there exists a truth assignment v such that $v \models \varphi(x)$. The truth assignment tells, for every node labeled with a Boolean test, whether that test passes or not. In our example, x_{a_3} is reachable when both the tests b_1 and b_2 fail; this is exactly what satisfiability of $\varphi(x_{a_3})$ entails (viz., that both b_1 and b_2 must be false to satisfy $\neg x_{b_1} \wedge \neg x_{b_2}$).
- This intuition justifies the other two requirements of Definition 3.1(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.

Next, we define an order on LPOFs, which we will use for the construction of fixed points to give semantics to unbounded loops. The intuition is that $\alpha \sqsubseteq_{\mathsf{lpof}} \beta$ iff β has *more behaviors* than α, meaning that β can be obtained by expanding $\bot$ nodes in α into larger structures (each unfolding of a while-loop is obtained in this way, as shown in Section 5.1). Since LPOFs can only be expanded from $\bot$ nodes, the label set L must be pointed (*i.e.,* $\bot \in L$); however, the order on L need not be flat: labels themselves can also become larger when passing from α to β (*e.g.,* if actions are nondeterministic assignments from a given set, this corresponds to enlarging the set of possible choices). This can also be useful to model *invariant sensitive execution*, introduced in Probabilistic Concurrent Outcome Logic [70]. Notationally, Bot_α denotes the set of nodes labelled with $\bot$, *i.e.,* $\mathsf{Bot}_\alpha \triangleq \{x \in N_\alpha \mid \lambda_\alpha(x) = \bot\}$.

▶ **Definition 3.2** (Ordering on LPOFs). *We let $\alpha \sqsubseteq_{\mathsf{lpof}} \beta$ iff:*

1. *N_α is a downward-closed subset of N_β, written $N_\alpha \subseteq_\downarrow N_\beta$;*

2. *$<_\alpha = <_\beta \cap (N_\alpha \times N_\alpha)$;*

3. *$\forall x \in N_\alpha$:*
   - **a.** *$\lambda_\alpha(x) \leq \lambda_\beta(x)$;*
   - **b.** *$\varphi_\alpha(x) = \varphi_\beta(x)$;*
   - **c.** *$\mathsf{succ}_\alpha(x) = \mathsf{succ}_\beta(x) \setminus \mathsf{Bot}_\alpha{\uparrow}_\beta$.*

▶ **Example 3.3.** Consider the following three LPOFs:

$$\alpha = \begin{tikzcd} y_1(\bot) & & y_2 \\ & x \arrow[ul] \arrow[ur] & \end{tikzcd} \qquad \beta = \begin{tikzcd} & & z \\ y_1(\bot) & & y_2 \arrow[u] \\ & x \arrow[ul] \arrow[ur] & \end{tikzcd} \qquad \gamma = \begin{tikzcd} & z & \\ y_1 \arrow[ur] & & y_2 \arrow[ul] \\ & x \arrow[ul] \arrow[ur] & \end{tikzcd}$$

where node y_1 is labelled with $\bot$ in α and β, whereas all other nodes have non-$\bot$ labels. Following the intuition given above, both α and β are attempts to specify that the computation in γ is truncated at y_1. However, only $\alpha \sqsubseteq_{lpof} \gamma$; this is because, if $\bot$ represents a diverging computation, then z will never have all the causes it needs to be executed and so it must disappear. Indeed, $\beta \not\sqsubseteq_{lpof} \gamma$ because, even though $N_\beta \subseteq_\downarrow N_\gamma$, condition (3c) of Definition 3.2 is violated: $z \in \mathsf{succ}_\beta(y_2)$ but $z \notin \mathsf{succ}_\gamma(y_2) \setminus \mathsf{Bot}_\beta{\uparrow}_\gamma$, since $y_1 \in \mathsf{Bot}_\beta$ and $y_1 <_\gamma z$.

Finally, in order to find fixed points, we will need to have a directed complete partial order (DCPO) structure and a notion of continuity for functions, which we now define.

▶ **Definition 3.4** (DCPO). *Given a poset $\langle X, \leq \rangle$, a subset $D \subseteq X$ is called* directed *iff it is not empty and, for every two elements $x_1, x_2 \in D$, there exists $x \in D$ such that $x_1, x_2 \leq x$.*

$\langle X, \leq \rangle$ is a directed complete partial order *(DCPO) iff, for every directed set D, $\sup D$ exists; furthermore, $\langle X, \leq \rangle$ is called* pointed *if there exists an element $\bot \in X$ such that $\bot \leq x$ for all $x \in X$.*

▶ **Definition 3.5** (Scott Continuity). *Given two DCPOs $\langle X, \leq_X \rangle$ and $\langle Y, \leq_Y \rangle$, a function $f \colon X \to Y$ is* Scott Continuous *if it is monotone: $f(x) \leq_Y f(x')$ for all $x \leq_X x'$; and preserves suprema of directed sets: $\sup_{x \in D} f(x) = f(\sup D)$ for all $D \subseteq X$ directed.*

▶ **Lemma 3.6.** *For any directed set $D \subseteq \mathit{lpof}(L)$, $\sup D = \langle N, <, \lambda, \varphi \rangle$, where:*

$$N \triangleq \bigcup_{\beta \in D} N_\beta \qquad < \triangleq \bigcup_{\beta \in D} <_\beta \qquad \lambda(x) \triangleq \sup_{\beta \in D \,:\, x \in N_\beta} \lambda_\beta(x) \qquad \varphi(x) \triangleq \psi_x$$

and $\psi_x = \varphi_\beta(x)$, for all $\beta \in D$ such that $x \in N_\beta$.

In Lemma 3.6, φ is well defined since, for every $x \in \mathit{nodes}$ and $\beta, \beta' \in D$ that contain x, it must be that $\varphi_\beta(x) = \varphi_{\beta'}(x)$. Indeed, since D is directed, there must exist a $\gamma \in D$ such that $\beta, \beta' \sqsubseteq_{lpof} \gamma$ and, by Definition 3.2(3b), we have that $\varphi_\beta(x) = \varphi_\gamma(x) = \varphi_{\beta'}(x)$.

Notice that $\mathit{lpof}(L)$ is *not* pointed, since the node at the root can vary; hence, there is no single LPOF that is smaller (w.r.t. $\sqsubseteq_{lpof}$) than all other LPOFs. This situation will change when moving to pomsets [67, Lemma D.6], which abstract away from the specific nodes.

3.2 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 used.

▶ **Definition 3.7** (LPOF Isomorphism). *Let α and β be LPOFs and $f \colon N_\alpha \to N_\beta$ be a bijection between their nodes. Define $f(\alpha) = \langle N, <, \lambda, \varphi \rangle$ as*

$$N \triangleq \{f(x) \mid x \in N_\alpha\} \quad < \triangleq \{(f(x), f(y)) \mid x <_\alpha y\} \quad \lambda \triangleq \lambda_\alpha \circ f^{-1} \quad \varphi \triangleq f \circ \varphi \circ f^{-1}$$

where $f(\psi)$ syntactically renames the variables of ψ in the obvious way; α and β are isomorphic, written $\alpha \equiv \beta$, iff there is a bijection $f \colon N_\alpha \to N_\beta$ such that $f(\alpha) = \beta$.

▶ **Definition 3.8** (Pomsets with Formulae). *Let $[\alpha] \triangleq \{\beta \in \mathit{lpof}(L) \mid \alpha \equiv \beta\}$ be the isomorphism class of α. A pomset with formulae (or, simply, pomset) $\boldsymbol{\alpha} \in \mathit{pom}(L)$ is an isomorphism class of* LPOFs: *$\mathit{pom}(L) \triangleq \{[\alpha] \mid \alpha \in \mathit{lpof}(L)\}$. Let $\mathit{pom}_{\mathsf{fin}}(L)$ be the set of pomsets with finitely many nodes. Finally, let $\boldsymbol{\alpha} \sqsubseteq_{\mathit{pom}} \boldsymbol{\beta}$ iff $\forall \alpha \in \boldsymbol{\alpha}. \exists \beta \in \boldsymbol{\beta}.\ \alpha \sqsubseteq_{\mathit{lpof}} \beta$.*

The semantics of programs depends on a variety of pomset composition operations (sequential, guarded, and parallel composition). We will also later *linearize* pomsets into state transformer functions. Some of these operations cannot be defined inductively on infinite structures, so we need ways to extend functions on finite structures to infinite ones. We first show that infinite pomsets correspond to the suprema of their finite approximations.

▶ **Lemma 3.9** (Finite Approximations). *For any $\boldsymbol{\alpha} \in \mathit{pom}(L)$, $\boldsymbol{\alpha} = \sup \lfloor \boldsymbol{\alpha} \rfloor_{\mathsf{fin}}$, where $\lfloor \boldsymbol{\alpha} \rfloor_{\mathsf{fin}} \triangleq \{\boldsymbol{\beta} \in \mathit{pom}_{\mathsf{fin}}(L) \mid \boldsymbol{\beta} \sqsubseteq_{\mathit{pom}} \boldsymbol{\alpha}\}$ is the set of finite approximations of $\boldsymbol{\alpha}$.*

Finite approximations coincide with the approximation order $\ll$ of [1] instantiated to pomsets (see [67, Appendix E] for more details); thus, from now on, we let $\boldsymbol{\alpha} \ll \boldsymbol{\beta}$ denote $\boldsymbol{\alpha} \in \lfloor \boldsymbol{\beta} \rfloor_{\mathsf{fin}}$. We are now ready to show that every operation f on finite pomsets can be extended to infinite ones by defining the operation on $\boldsymbol{\alpha}$ as the sup of the images through f of all the approximations of $\boldsymbol{\alpha}$. This will be useful in the remainder of the paper.

▶ **Lemma 3.10** (Extension). *Let $f \colon \mathit{pom}_{\mathsf{fin}}(L)^n \to T$ be a monotone function on the* DCPO *$\langle T, \leq \rangle$. Then $f^* \colon \mathit{pom}(L)^n \to T$, shown below, is well-defined and Scott continuous:*

$$f^*(\boldsymbol{\alpha}_1, \ldots, \boldsymbol{\alpha}_n) \triangleq \sup_{\boldsymbol{\alpha}'_1 \ll \boldsymbol{\alpha}_1} \cdots \sup_{\boldsymbol{\alpha}'_n \ll \boldsymbol{\alpha}_n} f(\boldsymbol{\alpha}'_1, \ldots, \boldsymbol{\alpha}'_n)$$

4 Pomset Operations

In this section, we define pomset operations that mirror the program syntax $C \in \mathit{cmd}$ from (1). We first define the singleton LPOF $\langle \ell \rangle_x$ on node x with label $\ell \in L$; this generalizes to $\langle \ell \rangle$ on pomsets and will be used to give the semantics to **skip** and atomic actions:

$$\langle \ell \rangle_x \triangleq \langle \{x\}, \emptyset, [x \mapsto \ell], [x \mapsto \mathsf{true}] \rangle \in \mathit{lpof}(L) \qquad \langle \ell \rangle \triangleq \{\langle \ell \rangle_x \mid x \in \mathit{nodes}\} \in \mathit{pom}(L)$$

4.1 Guarded Choice

We now define guard, recording the causality between a test and the two branches of computation defined on it. Assuming that $N_\alpha \cap N_\beta = \emptyset$ and $x \notin N_\alpha \cup N_\beta$, we let $\mathsf{guard}(x, \ell, \alpha, \beta) = \langle N, <, \lambda, \varphi \rangle$ where we overload $\wedge$ s.t. $\psi \wedge \mathsf{true} = \mathsf{true} \wedge \psi = \psi$, and:

$$N \triangleq \{x\} \cup N_\alpha \cup N_\beta \qquad\qquad < \triangleq <_\alpha \cup <_\beta \cup (\{x\} \times (N_\alpha \cup N_\beta))$$

$$\lambda(y) \triangleq \begin{cases} \ell & \text{if } y = x \\ \lambda_\alpha(y) & \text{if } y \in N_\alpha \\ \lambda_\beta(y) & \text{if } y \in N_\beta \end{cases} \qquad\qquad \varphi(y) \triangleq \begin{cases} \mathsf{true} & \text{if } y = x \\ \varphi_\alpha(y) \wedge x & \text{if } y \in N_\alpha \\ \varphi_\beta(y) \wedge \neg x & \text{if } y \in N_\beta \end{cases}$$

So, $\mathsf{guard}(x, b, \alpha, \beta)$ joins α and β with a new root node x, whose label is b, and additionally, the formulae in α and β are updated to require that b 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 (4). For finite pomsets, we define the guard operation as follows:

$$\mathsf{guard}(\ell, \boldsymbol{\alpha}, \boldsymbol{\beta}) \triangleq \{\mathsf{guard}(x, \ell, \alpha, \beta) \mid \alpha \in \boldsymbol{\alpha},\ \beta \in \boldsymbol{\beta}, N_\alpha \cap N_\beta = \emptyset, x \notin N_\alpha \cup N_\beta\}$$

Since guard is monotone [67, Lemma F.8], we can extend it to infinite pomsets:

$$\mathsf{guard}(\ell, \boldsymbol{\alpha}, \boldsymbol{\beta}) \triangleq \sup_{\boldsymbol{\alpha}' \ll \boldsymbol{\alpha}} \sup_{\boldsymbol{\beta}' \ll \boldsymbol{\beta}} \mathsf{guard}(\ell, \boldsymbol{\alpha}', \boldsymbol{\beta}')$$

4.2 Sequential Composition

The sequential composition operation $\boldsymbol{\alpha} \fatsemi \boldsymbol{\beta}$ is meant to enforce that all actions in $\boldsymbol{\alpha}$ must occur before any of those in $\boldsymbol{\beta}$. This causality dependency interacts with guarded branching, as we will now see, which makes the formal definition of $\fatsemi$ more challenging. Consider the following LPOFs (where we denote with x the singleton $\langle \ell \rangle_x$ when the label is not relevant):

$$\text{a. } x \fatsemi y = \begin{tikzcd} y \\ x \arrow[u] \end{tikzcd} \qquad \text{b. } \left(\begin{tikzcd} y_1 & & y_2 \\ & x \arrow[ul] \arrow[ur] & \end{tikzcd}\right) \fatsemi z = \begin{tikzcd} & z & \\ y_1 \arrow[ur] & & y_2 \arrow[ul] \\ & x \arrow[ul] \arrow[ur] & \end{tikzcd} \qquad \text{c. } \left(\begin{tikzcd} y_1 & & y_2 \\ & x \arrow[ul, "\mathsf{T}"'] \arrow[ur, "\mathsf{F}"] & \end{tikzcd}\right) \fatsemi z = \begin{tikzcd} z_x & & z_{\neg x} \\ y_1 \arrow[u] & & y_2 \arrow[u] \\ & x \arrow[ul, "\mathsf{T}"'] \arrow[ur, "\mathsf{F}"] & \end{tikzcd} \quad (5)$$

The sequential composition of two singletons (5a) simply creates a causality between them. When the program has forked into parallel (unguarded) branches (5b), $\fatsemi$ introduces a diamond structure, as is typical in pomset semantics. However, if the program has a *guarded* branch (5c), 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, invalidating Lemma 3.9; see Figure 2 and [67, Appendix B.2] for a more detailed account. When a structure contains $\bot$ nodes, so that some paths are *stuck*, the behavior of $\fatsemi$ is more complicated. Consider the following LPOFs, where node y_1 is labelled with $\bot$ and all other nodes have non-$\bot$ labels:

$$\text{a. } \left(\begin{tikzcd} y_1(\bot) & & y_2 \\ & x \arrow[ul, "\mathsf{T}"'] \arrow[ur, "\mathsf{F}"] & \end{tikzcd}\right) \fatsemi z = \begin{tikzcd} & & z \\ y_1(\bot) & & y_2 \arrow[u] \\ & x \arrow[ul, "\mathsf{T}"'] \arrow[ur, "\mathsf{F}"] & \end{tikzcd} \qquad \text{b. } \left(\begin{tikzcd} y_1(\bot) & & y_2 \\ & x \arrow[ul] \arrow[ur] & \end{tikzcd}\right) \fatsemi z = \begin{tikzcd} y_1(\bot) & & y_2 \\ & x \arrow[ul] \arrow[ur] & \end{tikzcd} \quad (6)$$

When composing the singleton z after a guarded branch with one stuck path (6a), 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 at the test x is false. However, the same is not true for parallel branching; in (6b), 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. This behavior ensures monotonicity of $\fatsemi$ [67, Lemma F.5].

We first define $\fatsemi$ for finite LPOFs, and then extend our definition to infinite ones using Lemma 3.10. In the rest of this section, we assume that $\alpha, \beta \in \mathit{lpof}_{\mathsf{fin}}(L)$. We start by defining the notion of stuck computations, extensible nodes, and branches.

$$\mathsf{stuck}_\alpha \triangleq \bigvee_{x \in \mathsf{Bot}_\alpha} \varphi_\alpha(x) \qquad \mathsf{br}_\alpha \triangleq \left\{ \varphi_\alpha(S) \;\middle|\; \begin{array}{l} \emptyset \subset S \subseteq \mathsf{ext}_\alpha,\ \varphi_\alpha(S) \Rightarrow \neg\mathsf{stuck}_\alpha, \\ \forall T.\ S \subset T \subseteq \mathsf{ext}_\alpha \Rightarrow \neg\mathsf{sat}(\varphi_\alpha(T)) \end{array} \right\}$$
$$\mathsf{ext}_\alpha \triangleq \{x \in N_\alpha \mid \varphi_\alpha(x) \not\Rightarrow \mathsf{stuck}_\alpha\}$$

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. In (5), all nodes are extensible, whereas in (6a) only x and y_2 are extensible and in (6b) 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: there, for a set $S \subseteq N_\alpha$, we define $\varphi_\alpha(S) \triangleq \bigwedge_{x \in S} \varphi_\alpha(x)$. For example, in (5a,b), $\mathsf{br} = \{\mathsf{true}\}$ since there are no tests; in (5c), $\mathsf{br} = \{x, \neg x\}$ since both outcomes of the test are not stuck; in (6a), $\mathsf{br} = \{\neg x\}$ since the program is stuck if x passes; and in (6b), $\mathsf{br} = \emptyset$, since all paths are stuck. The set of branches increases monotonically with $\sqsubseteq_{\mathit{lpof}}$. Refer to [67, Appendix B.1] for more examples of extensibility and branches. As we saw in (5c), we will need an isomorphic copy $\beta_\psi \equiv \beta$ for each branch $\psi \in \mathsf{br}_\alpha$. These copies are generated by a function $f\colon \mathsf{br}_\alpha \to [\beta]$ 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:

$$\mathsf{copy}_{\alpha,\beta} \triangleq \left\{ f\colon \mathsf{br}_\alpha \to [\beta] \;\middle|\; \forall \psi \in \mathsf{br}_\alpha.\, (N_{f(\psi)} \cap N_\alpha = \emptyset \;\wedge\; \forall \psi' \neq \psi. N_{f(\psi)} \cap N_{f(\psi')} = \emptyset) \right\}$$

Given any $\alpha, \beta \in \mathit{lpof}_{\mathsf{fin}}(L)$ and $f \in \mathsf{copy}_{\alpha,\beta}$, we define sequential composition as $\alpha \fatsemi_f \beta = \langle N, <, \lambda, \varphi \rangle$ where $\beta_\psi \triangleq f(\psi)$ and the components are defined as follows:

$$N \triangleq N_\alpha \cup \bigcup_{\psi \in \mathsf{br}_\alpha} N_{\beta_\psi} \qquad\qquad < \triangleq <_\alpha \cup \bigcup_{\psi \in \mathsf{br}_\alpha} \left(<_{\beta_\psi} \cup (\{ x \in N_\alpha \mid \psi \Rightarrow \varphi_\alpha(x) \} \times N_{\beta_\psi}) \right)$$

$$\lambda(x) \triangleq \begin{cases} \lambda_\alpha(x) & \text{if } x \in N_\alpha \\ \lambda_{\beta_\psi}(x) & \text{if } x \in N_{\beta_\psi} \end{cases} \qquad \varphi(x) \triangleq \begin{cases} \varphi_\alpha(x) & \text{if } x \in N_\alpha \\ \varphi_{\beta_\psi}(x) \wedge \psi & \text{if } x \in N_{\beta_\psi} \end{cases}$$

(where, again, $\psi \wedge \mathsf{true} = \mathsf{true} \wedge \psi = \psi$). 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. All labels are preserved, and formulae in β_ψ are updated to also include ψ. Sequential composition for finite pomsets is defined below (left), and is defined for infinite pomsets by extension (right).

$$\boldsymbol{\alpha} \fatsemi \boldsymbol{\beta} \triangleq \{ \alpha \fatsemi_f \beta \mid \alpha \in \boldsymbol{\alpha},\ \beta \in \boldsymbol{\beta},\ f \in \mathsf{copy}_{\alpha,\beta} \} \qquad \boldsymbol{\alpha} \fatsemi \boldsymbol{\beta} \triangleq \sup_{\boldsymbol{\alpha}' \ll \boldsymbol{\alpha}} \sup_{\boldsymbol{\beta}' \ll \boldsymbol{\beta}} \boldsymbol{\alpha}' \fatsemi \boldsymbol{\beta}'$$

▶ Remark 4.1. We can now explain Condition (2c) in Definition 3.1, requiring single-rootedness, which may seem strange in a framework with concurrency; indeed, the usual semantics of two commands put in parallel is obtained by taking the (disjoint) union of their pomsets (and this yields several possible minimum elements in the resulting pomset). However, having multi-rooted LPOFs would make sequential composition not monotone. For example, $y \sqsubseteq_{\mathit{lpof}} y\ z$ but $x \fatsemi y \triangleq \begin{array}{c} y \\ \uparrow \\ x \end{array} \not\sqsubseteq_{\mathit{lpof}} \begin{array}{ccc} y & & z \\ & \nwarrow\ \nearrow & \\ & x & \end{array} \triangleq x \fatsemi (y\ z)$. This is undesirable and easily fixed with the mild requirement of single-rootedness in Definition 3.1.

4.3 Parallel Composition

We finally define parallel composition. We assume that L contains a special (non-$\bot$) label fork that denotes thread forking and let $\mathsf{nFork}_\alpha \triangleq N_\alpha \setminus \{ x \in \min_\alpha \mid \lambda_\alpha(x) = \mathsf{fork} \}$ denote the set of nodes of α without its fork-minimal elements (if any). 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

$$N \triangleq \{x\} \cup \mathsf{nFork}_\alpha \cup \mathsf{nFork}_\beta$$

$$< \triangleq (\{x\} \times (\mathsf{nFork}_\alpha \cup \mathsf{nFork}_\beta)) \cup (<_\alpha \cap (\mathsf{nFork}_\alpha \times \mathsf{nFork}_\alpha)) \cup (<_\beta \cap (\mathsf{nFork}_\beta \times \mathsf{nFork}_\beta))$$

$$\lambda(y) \triangleq \begin{cases} \mathsf{fork} & \text{if } y = x \\ \lambda_\alpha(y) & \text{if } y \in \mathsf{nFork}_\alpha \\ \lambda_\beta(y) & \text{if } y \in \mathsf{nFork}_\beta \end{cases} \qquad \varphi(y) \triangleq \begin{cases} \mathsf{true} & \text{if } y = x \\ \varphi_\alpha(y) & \text{if } y \in \mathsf{nFork}_\alpha \\ \varphi_\beta(y) & \text{if } y \in \mathsf{nFork}_\beta \end{cases}$$

Notice that the branching that arises from $\parallel$ is conceptually (and practically) different from the branching that arises from a test: the branching due to $\parallel$ does not exclude any branch, whereas the two branches due to a test are mutually exclusive. This is apparent by the different way in which we handle the formulae associated to the nodes after the branch: they remain the same under parallel composition, whereas they are extended with a new conjunct (specifying the value of the test at the branch) under a guard. The complication arising in our handling of parallel composition through fork is that we want one single node labelled with fork, 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 fork 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:

$$\alpha = \begin{tikzcd} y_1 & & y_2 \\ & x_1(\mathsf{fork}) \arrow[ul] \arrow[ur] & \end{tikzcd} \qquad \beta = \begin{tikzcd} z_1 & z_2 & z_3 \\ & x_2(\mathsf{fork}) \arrow[ul] \arrow[u] \arrow[ur] & \end{tikzcd} \qquad \alpha \parallel_x \beta = \begin{tikzcd} y_1 & y_2 & z_1 & z_2 & z_3 \\ & & x(\mathsf{fork}) \arrow[ull] \arrow[ul] \arrow[u] \arrow[ur] \arrow[urr] & & \end{tikzcd}$$

where x_1 and x_2 disappear and the new root is x, labelled fork.

As a side note, observe that the way in which we defined parallel composition entails that $\alpha \sqsubseteq_{lpof} \alpha \parallel_x \beta$ 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 fork that precedes a node labelled with $\bot$ and all the nodes of β. Also, a somewhat unusual notion of associativity for parallel of LPOFs holds, viz. $(\alpha \parallel_x \beta) \parallel_y \gamma = \alpha \parallel_y (\beta \parallel_z \gamma)$; the expectable notion of associativity is recovered for pomsets, where we abstract from the specific node set[3]. Like before, we define parallel composition for (both finite and infinite) pomsets on top of that for LPOFs:

$$\boldsymbol{\alpha} \parallel \boldsymbol{\beta} \triangleq \{\alpha \parallel_x \beta \mid \alpha \in \boldsymbol{\alpha},\ \beta \in \boldsymbol{\beta},\ x \notin N_\alpha \cup N_\beta\}$$

▶ Remark 4.2. It is impossible for $\parallel$ to be *both* associative *and* Scott continuous. If it were continuous, then $f(\boldsymbol{\alpha}) \triangleq \langle\ell\rangle \parallel \boldsymbol{\alpha}$ would have a least fixed point $\mathsf{lfp}(f) = \langle\ell\rangle \parallel (\langle\ell\rangle \parallel \cdots)$, and, due to associativity, this pomset is infinitely branching, which invalidates Lemma 3.9.

5 Denotational Semantics

We now define denotational semantics for commands $C \in \mathit{cmd}$ (see (1)) based on pom. Although actions at this stage are uninterpreted, pom allows for actions with a variety of different *computational effects*, which we will see in Section 5.2. Those actions $a \in \mathit{act}$ are drawn from a DCPO $\langle \mathit{act}, \leq_{\mathit{act}}\rangle$ and tests $b \in \mathit{test}$ are drawn from a set test (with a flat order), which is closed under conjunction, disjunction, and negation. We first define a compositional denotational model for this language in Section 5.1, and then in Section 5.2 we show how to interpret pomsets as state transformers, to learn their behavior on particular inputs.

5.1 Pomset Semantics

We will now discuss the semantics of programs $C \in \mathit{cmd}$ as pomsets, which record both the causality between atomic actions and the branching behavior of tests. The label set $\mathit{label} \triangleq \mathit{act} \cup \mathit{test} \cup \{\mathsf{fork}, \bot\}$ consists of actions, tests, fork, and bottom nodes, which forms a pointed DCPO, with $\bot$ as the bottom. Actions are ordered according to $\leq_{\mathit{act}}$; for any other label $\ell \in \mathit{test} \cup \{\mathsf{fork}\}$, we only have $\bot \leq \ell$ and $\ell \leq \ell$. We will also henceforth use $\mathit{pom} \triangleq \mathit{pom}(\mathit{label})$ to refer to pomsets over this label set.

The semantic function $[\![-]\!] : \mathit{cmd} \to \mathit{pom}$, mapping commands to pomsets, is shown on the left of Figure 2. Its definition is straightforward using the pomset operations; in particular, **skip** is interpreted as a singleton fork and while-loops are interpreted as the least fixed point of the characteristic function $\Phi_{\langle C,b\rangle} : \mathit{pom} \to \mathit{pom}$ defined as follows:

$$\Phi_{\langle C,b\rangle}(\boldsymbol{\alpha}) \triangleq \mathsf{guard}(b, [\![C]\!] \fatsemi \boldsymbol{\alpha}, [\![\mathbf{skip}]\!])$$

[3] We remark that $\fatsemi$ on LPOFs also satisfies a non-straightforward associativity, that however becomes the usual one when passing to pomsets.

$$\begin{aligned}
[\![\textsf{skip}]\!] &\triangleq \langle \textsf{fork} \rangle \\
[\![a]\!] &\triangleq \langle a \rangle \\
[\![C_1 ; C_2]\!] &\triangleq [\![C_1]\!] \fatsemi [\![C_2]\!] \\
[\![C_1 \mid C_2]\!] &\triangleq [\![C_1]\!] \parallel [\![C_2]\!] \\
[\![\textbf{if } b \textbf{ then } C_1 \textbf{ else } C_2]\!] &\triangleq \textsf{guard}(b, [\![C_1]\!], [\![C_2]\!]) \\
[\![\textbf{while } b \textbf{ do } C]\!] &\triangleq \textsf{lfp}\left(\Phi_{\langle C,b\rangle}\right)
\end{aligned}$$

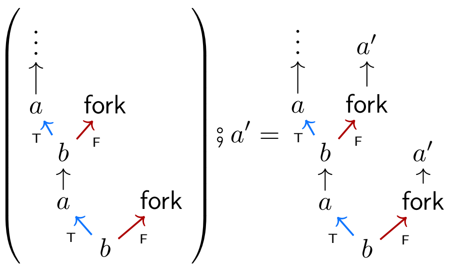
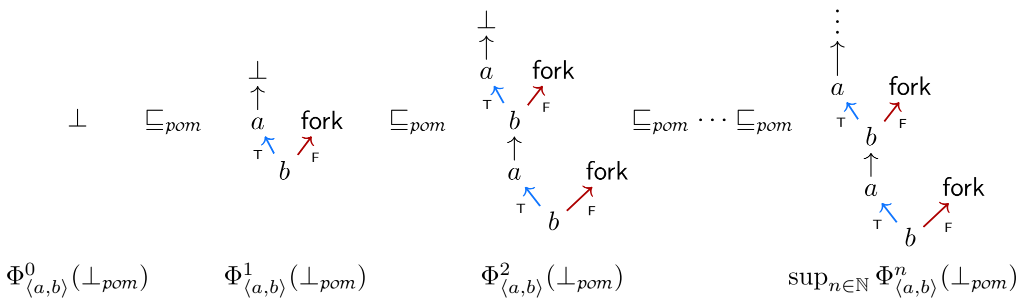


Figure 2 Left: pomset semantics of commands $[\![-]\!] : \mathit{cmd} \to \mathit{pom}$. Right: semantics of the program (**while** b **do** a) ; a'.

Clearly $\Phi_{\langle C,b\rangle}$ is Scott continuous, as it is defined by Scott continuous operations $\textsf{guard}$ and $\fatsemi$ [67, Corollaries F.6 and F.9]. So, by Kleene's fixed point theorem, $\Phi_{\langle C,b\rangle}$ has a least fixed point given by:

$$[\![\textbf{while } b \textbf{ do } C]\!] = \textsf{lfp}\left(\Phi_{\langle C,b\rangle}\right) = \sup_{n\in\mathbb{N}} \Phi^n_{\langle C,b\rangle}(\bot_{\mathit{pom}})$$

where $f^0 \triangleq \textsf{id}$ and $f^{n+1} \triangleq f \circ f^n$. Considering a loop $[\![\textbf{while } b \textbf{ do } a]\!]$ with a single action a as its body, we can unroll the definition several times to get the following infinite chain:



The supremum of this chain 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.

On the right of Figure 2, we show $[\![(\textbf{while } b \textbf{ do } a) \ ; a']\!]$, containing a loop with a single action in the body followed by another action. As discussed in Section 4.2, sequentially composing another action a' after a loop produces a copy of a' for each branch, resulting in countably many copies of a'. This approach is necessary to make a' finitely preceded.

The semantics obeys a binary branching property, guaranteeing that test nodes branch into exactly two successors, one where the test passes and another where it fails. Further, variables can only be introduced into formulae after a test. From now on, we assume that all pomsets have this property, which is clearly preserved by the operations of Section 4.

▶ **Definition 5.1** (Binary Branching). *A pomset $\boldsymbol{\alpha} \in \mathit{pom}$ has the* binary branching property *if, for every $\alpha \in \boldsymbol{\alpha}$ and every $x \in N_\alpha$ such that $\lambda_\alpha(x) \in \mathit{test}$, $\textsf{succ}_\alpha(x) = \{y_1, y_2\}$ such that:*

1. $\varphi_\alpha(y_1) \Leftrightarrow \varphi_\alpha(x) \wedge x$;
2. $\varphi_\alpha(y_2) \Leftrightarrow \varphi_\alpha(x) \wedge \neg x$; *and*
3. $\textsf{pred}_\alpha(y_1) = \textsf{pred}_\alpha(y_2) = \{x\}$.

Furthermore, if x is not the successor of a test, then $\varphi_\alpha(x) \Leftrightarrow \bigwedge_{y\in\textsf{pred}_\alpha(x)} \varphi_\alpha(y)$.

$$\mathsf{next}(\alpha,\psi,S) \triangleq \{x \in N_\alpha \setminus S \mid x{\downarrow_\alpha} \subseteq S, \psi \Rightarrow \varphi_\alpha(x)\}$$

$$\mathcal{L}_{\mathit{lpof}}(\alpha,\psi,S)(s) \triangleq \eta(s) \quad \text{if } \mathsf{next}(\alpha,\psi,S)=\emptyset$$

$$\mathcal{L}_{\mathit{lpof}}(\alpha,\psi,S)(s) \triangleq \mathop{\&}_{x\in\mathsf{next}(\alpha,\psi,S)} \begin{cases} \mathcal{L}_{\mathit{lpof}}\left(\alpha,\psi,S\cup\{x\}\right)^\dagger\left(\llbracket a\rrbracket_{\mathit{act}}(s)\right) & \text{if } \lambda_\alpha(x)=a\in\mathit{act} \\ \mathcal{L}_{\mathit{lpof}}(\alpha,\psi\wedge(\!|x=\llbracket b\rrbracket_{\mathit{test}}(s)|\!), S\cup\{x\})(s) & \text{if } \lambda_\alpha(x)=b\in\mathit{test} \\ \bot_D & \text{if } \lambda_\alpha(x)=\bot \\ \mathcal{L}_{\mathit{lpof}}(\alpha,\psi,S\cup\{x\})(s) & \text{if } \lambda_\alpha(x)=\mathsf{fork} \end{cases}$$

Figure 3 Linearization for finite LPOFs $\mathcal{L}_{\mathit{lpof}}\colon \mathit{lpof}_{\mathsf{fin}}(\mathit{label})\times\mathit{form}\times\mathcal{P}(\mathit{nodes})\to\mathcal{S}\to D(\mathcal{S})$.

5.2 Linearization

We now discuss how to produce a state transformer from the pomset semantics of Section 5.1, which is useful for understanding the input-output behavior of programs resulting from scheduling concurrent threads. This state transformer is obtained via *linearization*, which consists of considering all interleavings of the threads in order to interpret the semantics of a *complete* program as a function from inputs to collections of outputs. Once the program is linearized, more threads cannot be composed in parallel, as the causality information is gone. Nevertheless, linearization is useful to understand the program's behavior, and relate the behavior to other semantic models, as shown in Sections 2 and 6.

When performing linearization, we will interpret programs in a computational domain D, which must support nondeterminism – in order to model the different ways that threads can be interleaved – but may support additional computational effects too, which we will model using monads. Hence, we assume some familiarity with basic notions of monads and Kleisli categories; for a detailed introduction, we refer to [3]. More precisely, we need the following:

1. A domain D and set of states $\mathcal{S}$, such that $\langle D(\mathcal{S}),\sqsubseteq\rangle$ is a pointed DCPO with bottom $\bot_D$.
2. An associative, commutative, and monotone nondeterminism operator $\&\colon D(\mathcal{S})^2\to D(\mathcal{S})$.
3. An additive monad in the category **dcpo** $\langle D,\eta,(-)^\dagger\rangle$, meaning that:
   a. the operations $\eta\colon X\to D(X)$ and $(-)^\dagger\colon (X\to D(Y))\to D(X)\to D(Y)$ obey the monad laws $f^\dagger\circ\eta=f$, $\eta^\dagger=\mathsf{id}$, and $(g^\dagger\circ f)^\dagger=g^\dagger\circ f^\dagger$.
   b. Kleisli extension is Scott Continuous (*i.e.*, $\sup_{f\in D}\sup_{d\in D'} f^\dagger(d)=(\sup D)^\dagger(\sup D')$), strict (*i.e.*, $f^\dagger(\bot_D)=\bot_D$), and additive (*i.e.*, $f^\dagger(d\,\&\,d')=f^\dagger(d)\,\&\,f^\dagger(d')$) [66,69].
4. An interpretation function for actions $\llbracket-\rrbracket_{\mathit{act}}:\mathit{act}\to\mathcal{S}\to D(\mathcal{S})$ that is monotone (*i.e.*, $\llbracket a\rrbracket_{\mathit{act}}(s)\sqsubseteq\llbracket a'\rrbracket_{\mathit{act}}(s)$ if $a\le_{\mathit{act}} a'$) and one for tests $\llbracket-\rrbracket_{\mathit{test}}:\mathit{test}\to\mathcal{S}\to\mathbb{B}$.

There are many domains that obey these properties, depending on which *computational effects* are present. These include: the Hoare, Smyth, and Plotkin powerdomains for nondeterministic computation [1,46,53] (where the latter two additionally deal with nontermination), the convex powerset [20,42] and the powerdomain of indexed valuations [60,61,63] (for mixed probabilistic and nondeterministic computation), and other domains for nondeterminism with exceptions [66,69]. In Theorems 5.3 and 6.1, we will show that standard semantics in two of those domains are recovered from our pomset semantics.

We will start by defining a linearization operation on finite LPOFs, which is shown in Figure 3. 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 we can extend linearization to infinite structures using the extension lemma. In $\mathcal{L}_{\mathit{lpof}}(\alpha,\psi,S)$, 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 (α, ψ, S) gives the nodes that are ready to be scheduled, which includes all nodes that obey the path condition ψ, and whose predecessors are in S. If (α, ψ, S) is empty, then $\mathcal{L}_{\mathit{lpof}}(\alpha, \psi, S)$ simply returns the current state s using the monad unit η. If not, then it nondeterministically selects a next node, where $\binampersand_{i\in I} d_i \triangleq d_{i_1} \mathbin{\&} \cdots \mathbin{\&} d_{i_n}$, for any finite index set $I = \{i_1, \ldots, i_n\}$. If the next node is an action, then it interprets the action using $[\![-]\!]_{\mathit{act}}$, and then 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 result is added to the path condition, where $(\!|x = 1|\!) \triangleq x$ and $(\!|x = 0|\!) \triangleq \neg x$. If the next node is $\bot$, then the linearization is $\bot_D$, and fork is treated like a no-op.

We next use $\mathcal{L}_{\mathit{lpof}}$ to define linearization on finite pomsets $\mathcal{L}_{\mathsf{fin}} \colon \mathit{pom}_{\mathsf{fin}}(\mathit{label}) \to \mathcal{S} \to D(\mathcal{S})$ as $\mathcal{L}_{\mathsf{fin}}([\alpha]) \triangleq \mathcal{L}_{\mathit{lpof}}(\alpha, \mathsf{true}, \emptyset)$. Clearly $\mathcal{L}_{\mathit{lpof}}(\alpha, \mathsf{true}, \emptyset) = \mathcal{L}_{\mathit{lpof}}(\beta, \mathsf{true}, \emptyset)$ if $\alpha \equiv \beta$, so $\mathcal{L}_{\mathsf{fin}}(\boldsymbol{\alpha})$ can be defined for any arbitrary representative LPOF $\alpha \in \boldsymbol{\alpha}$. Finally, linearization is extended to infinite pomsets $\mathcal{L} \colon \mathit{pom} \to \mathcal{S} \to D(\mathcal{S})$ by taking the supremum over all of the finite approximations: $\mathcal{L}(\boldsymbol{\alpha}) \triangleq \mathcal{L}^*_{\mathsf{fin}}(\boldsymbol{\alpha})$. Linearization over finite pomsets is monotone [67, Lemma G.3], therefore the extension is well-defined and Scott continuous.

To ensure that linearization acts as desired, we provide a sanity check lemma to relate the linearized semantics to well-known counterparts. Linearizing **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 $\Psi_{\langle f,b\rangle}$. The function $\Psi_{\langle f,b\rangle}$ inherits Scott continuity from the Kleisli composition operator, therefore the fixed point exists.

▶ **Lemma 5.2** (Linearization). *The following properties hold:*

$$\mathcal{L}([\![\textbf{skip}]\!]) = \eta \qquad \mathcal{L}([\![a]\!]) = [\![a]\!]_{\mathit{act}} \qquad \mathcal{L}([\![C_1; C_2]\!]) = \mathcal{L}([\![C_2]\!])^\dagger \circ \mathcal{L}([\![C_1]\!])$$

$$\mathcal{L}([\![\textbf{if } b \textbf{ then } C_1 \textbf{ else } C_2]\!])(s) = \begin{cases} \mathcal{L}([\![C_1]\!])(s) & \text{if } [\![b]\!]_{\mathit{test}}(s) = 1 \\ \mathcal{L}([\![C_2]\!])(s) & \text{if } [\![b]\!]_{\mathit{test}}(s) = 0 \end{cases}$$

$$\mathcal{L}([\![\textbf{while } b \textbf{ do } C]\!]) = \mathsf{lfp}\left(\Psi_{\langle \mathcal{L}([\![C]\!]), b\rangle}\right) \text{ where } \Psi_{\langle f,b\rangle}(g)(s) \triangleq \begin{cases} g^\dagger(f(s)) & \text{if } [\![b]\!]_{\mathit{test}}(s) = 1 \\ \eta(s) & \text{if } [\![b]\!]_{\mathit{test}}(s) = 0 \end{cases}$$

Lemma 5.2 is quite significant; it implies that $\mathcal{L}([\![C]\!])$ corresponds to standard monadic semantics in a variety of domains. In particular, it brings us to our first adequacy theorem, showing that our pomset model recovers standard probabilistic semantics $[\![C]\!]_{\mathcal{C}} : \mathcal{S} \to \mathcal{C}(\mathcal{S})$ (shown in Figure 4 of Section A.2) [20, 36, 68] based on the convex powerset for a parallel-free fragment of our programming language.

▶ **Theorem 5.3.** *For any program $C \in \mathit{cmd}$ without parallel composition: $\mathcal{L}([\![C]\!]) = [\![C]\!]_{\mathcal{C}}$.*

Proof. By induction on the structure of C. The cases follow immediately from Lemma 5.2. ◀

6 Pomset Languages and Purely Nondeterministic Concurrency

Powerdomains give semantics to programs that combine nondeterminism with looping or recursion [46, 53]. In this section, we create a powerdomain instantiation of our semantics from Section 5 and prove that the resulting model corresponds to well-known models of non-probabilistic concurrent programs [27, 34]. We work in the *Hoare* or *lower* powerdomain, named as such for its connection to partial correctness and Hoare Logic [16, 21]. Indeed, the

Hoare powerdomain identifies the terminating traces of a program, but not that the program *always* terminates. From a concurrency perspective, this corresponds to *safety* properties [33]. For simplicity, we presume that the order over actions $\leq_{act}$ in this section is flat.

Since typical pomsets do not contain formulae [17, 18, 48], they cannot encode control flow branching introduced by if-statements and while-loops. As we already saw in Section 2, concurrent semantics typically use *pomset languages* – sets of pomsets – where each individual pomset contains only the actions that occur in a single branch of computation [27, 34].

We emulate pomset language semantics using pomsets with formulae. First, we define a label set $\mathit{label}_{\mathsf{PL}} \triangleq \mathit{act}_{\mathsf{PL}} \cup \mathit{test}_{\mathsf{PL}} \cup \{\mathsf{fork}, \bot\}$, whose components are as before, but now $\mathit{test}_{\mathsf{PL}} \triangleq \emptyset$, and instead $\mathit{act}_{\mathsf{PL}} \triangleq \mathit{act} \cup \{\textbf{assume } b \mid b \in \mathit{test}\}$ includes an action **assume** b for every test $b \in \mathit{test}$, which guarantees that a certain condition holds in the current branch. Pomset languages $\mathit{pom\mathcal{L}ang} \triangleq \mathcal{P}(\mathit{pom}_{\mathsf{fin}}(\mathit{label}_{\mathsf{PL}}))$ are sets of finite pomsets over $\mathit{label}_{\mathsf{PL}}$.

The pomset language semantics $[\![-]\!]_{\mathsf{PL}} : \mathit{cmd} \to \mathit{pom\mathcal{L}ang}$, which is standard [27, 34], is given in Figure 5 of Section A.3. In this model, each $\boldsymbol{\alpha} \in [\![C]\!]_{\mathsf{PL}}$ corresponds to a particular sequence of test resolutions. When an **assume** fails, the trace is eliminated. When a branch occurs, *i.e.,* in an if-statement or while-loop, the set of traces is duplicated to account for both the "true" and "false" branches, as we saw in Figure 1. Loops are interpreted as the (possibly infinite) union of all finite traces, so infinite pomsets are not needed.

The Hoare powerdomain consists of sets of states, ordered by subset inclusion $\langle \mathcal{P}(\mathcal{S}), \subseteq \rangle$, which is known to be a pointed DCPO with supremum given by union $\cup$ and $\emptyset$ as bottom. Let $\& \triangleq \cup$ and define the monad operations as $\eta(s) \triangleq \{s\}$ and $f^\dagger(S) \triangleq \bigcup_{s \in S} f(s)$, which are well known to be Scott continuous, strict, and additive [66]. Accordingly, action evaluation $[\![-]\!]_{\mathit{act}_{\mathsf{PL}}} : \mathit{act}_{\mathsf{PL}} \to \mathcal{S} \to \mathcal{P}(\mathcal{S})$ is given below, where **assume** actions result in $\{s\}$, if the test is true in state s, and $\emptyset$, otherwise. We also define a specialized linearization operation $\mathcal{L}_{\mathsf{PL}} \colon \mathit{pom\mathcal{L}ang} \to \mathcal{S} \to \mathcal{P}(\mathcal{S})$, which is simply a union over the linearization of all traces.

$$[\![a]\!]_{\mathit{act}_{\mathsf{PL}}}(s) \triangleq \begin{cases} [\![a]\!]_{\mathit{act}}(s) & \text{if } a \in \mathit{act} \\ \{s\} & \text{if } a = \textbf{assume } b,\ [\![b]\!]_{\mathit{test}}(s) = 1 \\ \emptyset & \text{if } a = \textbf{assume } b,\ [\![b]\!]_{\mathit{test}}(s) = 0 \end{cases} \qquad \mathcal{L}_{\mathsf{PL}}(S)(s) \triangleq \bigcup_{\boldsymbol{\alpha} \in S} \mathcal{L}_{\mathsf{fin}}(\boldsymbol{\alpha})(s)$$

Since all formulae are true, the next nodes to schedule are exactly those whose predecessors have all been processed, *i.e.,* $(\alpha, \mathsf{true}, S) = \{x \in N_\alpha \setminus S \mid x{\downarrow_\alpha} \subseteq S\}$. This corresponds to standard interleaving definitions of linearization [48]. We now prove that pomset language semantics $[\![-]\!]_{\mathsf{PL}}$ (Figure 5) corresponds exactly to the Hoare powerdomain instance of our semantic model from Section 5. The proof relies on a translation $\mathsf{tr} \colon \mathit{pom} \to \mathit{pom\mathcal{L}ang}$, which recovers a pomset language from a pomset with formulae, and is defined in [67, Appendix H].

▶ **Theorem 6.1** (Equivalence of Semantics). *The following diagram commutes:*

$$\begin{tikzcd} \mathit{cmd} \arrow[r, "{[\![-]\!]}"] \arrow[d, "{[\![-]\!]_{\mathsf{PL}}}"'] & \mathit{pom} \arrow[d, "\mathcal{L}"] \arrow[dl, "\mathsf{tr}"'] \\ \mathit{pom\mathcal{L}ang} \arrow[r, "\mathcal{L}_{\mathsf{PL}}"'] & (\mathcal{S} \to \mathcal{P}(\mathcal{S})) \end{tikzcd}$$

The upper commuting triangle does not depend on the Hoare powerdomain, so it is tempting to say that pomset languages give an adequate model in other domains too. However, as we showed in Section 2, just because a semantic structure exists does not mean that it conveys the desired meaning. Indeed, the fact that $\mathcal{L} = \mathcal{L}_{\mathsf{PL}} \circ \mathsf{tr}$ relies on two particular properties of the Hoare powerdomain. First, if $h(x) = f(x) \,\&\, g(x)$, then $h^\dagger(S) = f^\dagger(S) \,\&\, g^\dagger(S)$, which is invalid in probabilistic domains. In fact, this corresponds to the problematic Equation (3) from Section 2. Second, $\mathcal{L}_{\mathsf{PL}}$ is computed as a union over a possibly infinite set (the union is infinite whenever the program contains a while-loop). Unbounded nondeterminism is known to cause problems in many domains – including the Smyth and Plotkin powerdomains [2,4,54] – thus no infinitary version of & exists.

So, although programs in any domain *can* be interpreted as pomset languages, those semantic objects may not give an adequate meaning to the program. Particularly in probabilistic domains – *i.e.,* the convex powerset – prior pomset language models [27,34] do not give us a way to find the relative probabilities of the different outcomes after actually running the program. Our new approach, using a single structure that records both causality and control flow, is more suitable for interpreting the semantics of concurrent programs in new domains, such as probabilistic computation. As we saw in Section 2 and Theorem 5.3, pomsets with formulae accurately capture the concurrent behavior of these programs, while also allowing us to recover the precise probabilities of each outcome.

7 Discussion and Related Work

We developed a new semantic structure – pomsets with formulae – and studied its domain-theoretic properties, which we used 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 [17,18,47], which have been used as a semantic basis for concurrent separation logic [8], concurrent Kleene algebras [22,26,27,34], and other applications [14,25,29].

While there are links between pomset semantics and operational models of concurrency [7], pomsets are often preferred for their finer notion of concurrency compared to the interleaving semantics offered by operational models. This is especially useful in the context of weak memory [25,29]. Probabilistic event structures model probabilistic process algebras [28,62,64,65], 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 computational effects in limited ways. For example, semantics for probabilistic concurrent programs have been defined in both operational [55,56] and denotational [70] styles, but these approaches are limited to programs with bounded looping constructs. The inclusion of unbounded looping adds significant complexity in the probabilistic case. Whereas finite-trace models are sufficient for many non-probabilistic scenarios, unbounded looping in probabilistic programs requires 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 [37,38] or metric [9] structure; prior work in this area was very informative to our own development, although the inclusion of deterministic branching in our own pomset structure added significant complexity.

An operational model of probabilistic concurrency has been developed based on Markov Decision Processes [15]. This semantics is used to lower bound expected values, but does not straightforwardly extend to more complex domains such as the convex powerset, which give the full set of distributions over outcomes. In addition, a denotational model has been developed [44], 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 capturing non-interleaving models of concurrency too.

Branching pomsets were recently introduced to model choices in choreographic programs [12–14]. 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 3.10) was necessary for linearization, without which we would not have been able to relate our pomset model to the known convex powerset semantics [20, 36, 68].

Going forward, it would be interesting to explore the applicability of pomsets with formulae to convex powerdomain constructions other than Smyth [30, 57–59], or to other domains for mixing probabilities and nondeterminism including indexed valuations [60, 61, 63] and multisets of distributions [24, 32]. While our linearization procedure is based on an *omniscient* scheduler, we are also interested in exploring more restricted models including oblivious schedulers (which cannot see the outcomes of random sampling), and fair schedulers. This will be challenging, as those models are non-compositional and therefore linearization on infinite pomsets could not be defined via Lemma 3.10. Still, pomsets with formulae are a good basis for studying these questions, as the structure itself makes no assumptions about how scheduling of parallel branches occurs.

References

1 Samson Abramsky and Achim Jung. *Domain theory*, pages 1–168. Oxford University Press, Inc., USA, 1995.

2 Krzysztof Apt and Gordon Plotkin. Countable nondeterminism and random assignment. *J. ACM*, 33(4):724–767, August 1986. doi:10.1145/6490.6494.

3 Steve Awodey. *Category Theory*. Oxford Logic Guides. Ebsco Publishing, 2006.

4 Ralph-Johan Back. Semantics of unbounded nondeterminism. In Jaco de Bakker and Jan van Leeuwen, editors, *Automata, Languages and Programming*, pages 51–63, Berlin, Heidelberg, 1980. Springer Berlin Heidelberg. doi:10.1007/3-540-10003-2_59.

5 Gilles Barthe, Thomas Espitau, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. An Assertion-Based Program Logic for Probabilistic Programs. In *Programming Languages and Systems*, pages 117–144, Cham, 2018. Springer International Publishing. doi:10.1007/978-3-319-89884-1_5.

6 Filippo Bonchi, Ana Sokolova, and Valeria Vignudelli. Presenting Convex Sets of Probability Distributions by Convex Semilattices and Unique Bases. In Fabio Gadducci and Alexandra Silva, editors, *9th Conference on Algebra and Coalgebra in Computer Science (CALCO 2021)*, volume 211 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 11:1–11:18, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CALCO.2021.11.

7 Stephen Brookes. Traces, Pomsets, Fairness and Full Abstraction for Communicating Processes. In Luboš Brim, Mojmír Křetínský, Antonín Kučera, and Petr Jančar, editors, *CONCUR 2002 — Concurrency Theory*, pages 466–482, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. doi:10.1007/3-540-45694-5_31.

8 Stephen Brookes. A semantics for concurrent separation logic. In Philippa Gardner and Nobuko Yoshida, editors, *CONCUR 2004 - Concurrency Theory*, pages 16–34, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. doi:10.1007/978-3-540-28644-8_2.

9 J. W. de Bakker and J. H. A. Warmerdam. Metric pomset semantics for a concurrent language with recursion. In Irène Guessarian, editor, *Semantics of Systems of Concurrent Processes*, pages 21–49, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg. doi:10.1007/3-540-53479-2_2.

10 Jerry den Hartog. Verifying Probabilistic Programs Using a Hoare like Logic. In P. S. Thiagarajan and Roland Yap, editors, *Advances in Computing Science — ASIAN'99*, pages 113–125, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. doi:10.1007/3-540-46674-6_11.

11 Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Inf.*, 1(2):115–138, June 1971. doi:10.1007/BF00289519.

12 Luc Edixhoven and Sung-Shik Jongmans. Realisability of Branching Pomsets. In Silvia Lizeth Tapia Tarifa and José Proença, editors, *Formal Aspects of Component Software*, pages 185–204, Cham, 2022. Springer International Publishing. doi:10.1007/978-3-031-20872-0_11.

13 Luc Edixhoven, Sung-Shik Jongmans, José Proença, and Ilaria Castellani. Branching pomsets: Design, expressiveness and applications to choreographies. *Journal of Logical and Algebraic Methods in Programming*, 136:100919, 2024. doi:10.1016/j.jlamp.2023.100919.

14 Luc Edixhoven, Sung-Shik Jongmans, José Proença, and Guillermina Cledou. Branching Pomsets for Choreographies. *Electronic Proceedings in Theoretical Computer Science*, 365:37–52, August 2022. doi:10.4204/eptcs.365.3.

15 Ira Fesefeldt, Joost-Pieter Katoen, and Thomas Noll. Towards Concurrent Quantitative Separation Logic. In Bartek Klin, Sławomir Lasota, and Anca Muscholl, editors, *33rd International Conference on Concurrency Theory (CONCUR 2022)*, volume 243 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:24, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CONCUR.2022.25.

16 Robert W. Floyd. Assigning Meanings to Programs. In *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32, Providence, Rhode Island, 1967. American Mathematical Society. doi:10.1007/978-94-011-1793-7_4.

17 Jay L. Gischer. The equational theory of pomsets. *Theoretical Computer Science*, 61(2):199–224, 1988. doi:10.1016/0304-3975(88)90124-7.

18 Jan Grabowski. On Partial Languages. *Fundamenta Informaticae*, 4(2):427–498, 1981. doi:10.3233/fi-1981-4210.

19 Sergiu Hart, Micha Sharir, and Amir Pnueli. Termination of probabilistic concurrent program. *ACM Trans. Program. Lang. Syst.*, 5(3):356–380, July 1983. doi:10.1145/2166.357214.

20 Jifeng He, Karen Seidel, and Annabelle McIver. Probabilistic models for the guarded command language. *Science of Computer Programming*, 28(2):171–192, 1997. Formal Specifications: Foundations, Methods, Tools and Applications. doi:10.1016/S0167-6423(96)00019-6.

21 Charles Antony Richard Hoare. An Axiomatic Basis for Computer Programming. *Commun. ACM*, 12(10):576–580, October 1969. doi:10.1145/363235.363259.

22 Tony Hoare, Bernhard Möller, Georg Struth, and Ian Wehrman. Concurrent Kleene Algebra and its Foundations. *J. Log. Algebraic Methods Program.*, 80(6):266–296, 2011. doi:10.1016/J.JLAP.2011.04.005.

23 Bart Jacobs. Coalgebraic trace semantics for combined possibilitistic and probabilistic systems. *Electronic Notes in Theoretical Computer Science*, 203(5):131–152, 2008. Proceedings of the Ninth Workshop on Coalgebraic Methods in Computer Science (CMCS 2008). doi:10.1016/j.entcs.2008.05.023.

24 Bart Jacobs. From Multisets over Distributions to Distributions over Multisets. In *Proceedings of the 36th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '21, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1109/LICS52264.2021.9470678.

25 Radha Jagadeesan, Alan Jeffrey, and James Riely. Pomsets with preconditions: a simple model of relaxed memory. *Proc. ACM Program. Lang.*, 4(Oopsla), November 2020. doi:10.1145/3428262.

26 Peter Jipsen and M. Andrew Moshier. Concurrent Kleene algebra with tests and branching automata. *Journal of Logical and Algebraic Methods in Programming*, 85(4):637–652, 2016. Relational and algebraic methods in computer science. doi:10.1016/j.jlamp.2015.12.005.

27 Tobias Kappé, Paul Brunet, Alexandra Silva, Jana Wagemaker, and Fabio Zanasi. *Concurrent Kleene Algebra with Observations: From Hypotheses to Completeness*, pages 381–400. Springer International Publishing, 2020. doi:10.1007/978-3-030-45231-5_20.

28 Joost-Pieter Katoen. *Quantitative and Qualitative Extensions of Event Structures*. Phd thesis - research ut, graduation ut, University of Twente, Netherlands, 1996. doi:10.3990/1.9789036507998.

29 Ryan Kavanagh and Stephen Brookes. A Denotational Semantics for SPARC TSO. *Electronic Notes in Theoretical Computer Science*, 336:223–239, 2018. The Thirty-third Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXIII). doi:10.1016/j.entcs.2018.03.025.

30 Klaus Keimel and Gordon Plotkin. Mixed powerdomains for probability and nondeterminism. *Logical Methods in Computer Science*, Volume 13, Issue 1, January 2017. doi:10.23638/LMCS-13(1:2)2017.

31 Dexter Kozen. Semantics of probabilistic programs. In *20th Annual Symposium on Foundations of Computer Science (SFCS '79)*, pages 101–114, 1979. doi:10.1109/SFCS.1979.38.

32 Dexter Kozen and Alexandra Silva. Multisets and distributions. In Venanzio Capretta, Robbert Krebbers, and Freek Wiedijk, editors, *Logics and Type Systems in Theory and Practice: Essays Dedicated to Herman Geuvers on The Occasion of His 60th Birthday*, pages 168–187, Cham, 2024. Springer Nature Switzerland. doi:10.1007/978-3-031-61716-4_11.

33 L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, 1977. doi:10.1109/TSE.1977.229904.

34 Michael R. Laurence and Georg Struth. Completeness Theorems for Bi-Kleene Algebras and Series-Parallel Rational Pomset Languages. In Peter Höfner, Peter Jipsen, Wolfram Kahl, and Martin Eric Müller, editors, *Relational and Algebraic Methods in Computer Science*, pages 65–82, Cham, 2014. Springer International Publishing. doi:10.1007/978-3-319-06251-8_5.

35 Daniel Lehmann and Michael O. Rabin. 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*, POPL '81, pages 133–138, New York, NY, USA, 1981. Association for Computing Machinery. doi:10.1145/567532.567547.

36 Annabelle McIver and Carroll Morgan. *Abstraction, Refinement and Proof for Probabilistic Systems.* Monographs in Computer Science. Springer, 2005. doi:10.1007/b138392.

37 J. J. Ch. Meyer and E. P. de Vink. Pomset semantics for true concurrency with synchronization and recursion. In Antoni Kreczmar and Grazyna Mirkowska, editors, *Mathematical Foundations of Computer Science 1989*, pages 360–369, Berlin, Heidelberg, 1989. Springer Berlin Heidelberg. doi:10.1007/3-540-51486-4_83.

38 J. J. Ch. Meyer and E.P. de Vink. Applications of compactness in the smyth powerdomain of streams. *Theoretical Computer Science*, 57(2):251–282, 1988. doi:10.1016/0304-3975(88)90042-4.

39 Michael Mislove. Nondeterminism and probabilistic choice: Obeying the laws. In Catuscia Palamidessi, editor, *CONCUR 2000 — Concurrency Theory*, pages 350–365, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg. doi:10.1007/3-540-44618-4_26.

40 Michael Mislove, Joël Ouaknine, and James Worrell. Axioms for probability and nondeterminism. *Electronic Notes in Theoretical Computer Science*, 96:7–28, 2004. Proceedings of the 10th International Workshop on Expressiveness in Concurrency. doi:10.1016/j.entcs.2004.04.019.

41 Carroll Morgan, Annabelle McIver, and Karen Seidel. Probabilistic Predicate Transformers. *ACM Trans. Program. Lang. Syst.*, 18(3):325–353, May 1996. doi:10.1145/229542.229547.

42 Carroll Morgan, Annabelle McIver, Karen Seidel, and J. W. Sanders. Refinement-oriented probability for CSP. *Form. Asp. Comput.*, 8(6):617–647, November 1996. doi:10.1007/bf01213492.

43 Robert Morris. Counting large numbers of events in small registers. *Commun. ACM*, 21(10):840–842, October 1978. doi:10.1145/359619.359627.

44 Renato Neves. An adequacy theorem between mixed powerdomains and probabilistic concurrency, 2024. doi:10.48550/arXiv.2409.15920.

45 Louis Parlant. *Monad Composition via Preservation of Algebras.* PhD thesis, University College London, 2020. URL: https://discovery.ucl.ac.uk/id/eprint/10112228/.

46 Gordon Plotkin. A powerdomain construction. *SIAM Journal on Computing*, 5(3):452–487, 1976. doi:10.1137/0205035.

47 Vaughan R. Pratt. Semantical Considerations on Floyd-Hoare Logic. In *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*, pages 109–121, 1976. doi:10.1109/SFCS.1976.27.

48 Vaughan R. Pratt. Modeling concurrency with partial orders. *Int. J. Parallel Program.*, 15(1):33–71, 1986. doi:10.1007/BF01379149.

49 Michael O. Rabin. N-process synchronization by 4.log2n-valued shared variable. In *21st Annual Symposium on Foundations of Computer Science (sfcs 1980)*, pages 407–410, 1980. doi:10.1109/SFCS.1980.26.

50 Michael O. Rabin. The choice coordination problem. *Acta Inf.*, 17(2):121–134, June 1982. doi:10.1007/BF00288965.

51 Dana Scott. Outline of a Mathematical Theory of Computation. Technical Report PRG02, OUCL, November 1970.

52 Dana Scott and Christopher Strachey. Toward A Mathematical Semantics For Computer Languages. Technical Report Prg06, OUCL, August 1971.

53 Michael Smyth. Power domains. *Journal of Computer and System Sciences*, 16(1):23–36, 1978. doi:10.1016/0022-0000(78)90048-X.

54 Harald Søndergaard and Peter Sestoft. Non-determinism in Functional Languages. *The Computer Journal*, 35(5):514–523, October 1992. doi:10.1093/comjnl/35.5.514.

55 Joseph Tassarotti. *Verifying Concurrent Randomized Algorithms*. PhD thesis, Carnegie Mellon University, December 2018. URL: https://csd.cmu.edu/academics/doctoral/degrees-conferred/joseph-tassarotti.

56 Joseph Tassarotti and Robert Harper. A Separation Logic for Concurrent Randomized Programs. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi:10.1145/3290377.

57 Regina Tix. *Continuous D-cones: convexity and powerdomain constructions*. PhD thesis, Darmstadt University of Technology, Germany, 1999. URL: https://d-nb.info/957239157.

58 Regina Tix. Convex power constructions for continuous d-cones. *Electronic Notes in Theoretical Computer Science*, 35:206–229, 2000. Workshop on Domains IV. doi:10.1016/S1571-0661(05)80746-7.

59 Regina Tix, Klaus Keimel, and Gordon Plotkin. Semantic domains for combining probability and non-determinism. *Electronic Notes in Theoretical Computer Science*, 222:3–99, 2009. doi:10.1016/j.entcs.2009.01.002.

60 Daniele Varacca. The powerdomain of indexed valuations. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 299–308, 2002. doi:10.1109/LICS.2002.1029838.

61 Daniele Varacca. *Probability, Nondeterminism and Concurrency: Two Denotational Models for Probabilistic Computation*. PhD thesis, University of Aarhus, November 2003. URL: https://www.brics.dk/DS/03/14/.

62 Daniele Varacca, Hagen Völzer, and Glynn Winskel. Probabilistic event structures and domains. *Theoretical Computer Science*, 358(2):173–199, 2006. Concurrency Theory (CONCUR 2004). doi:10.1016/j.tcs.2006.01.015.

63 Daniele Varacca and Glynn Winskel. Distributing probability over non-determinism. *Mathematical Structures in Computer Science*, 16(1):87–113, 2006. doi:10.1017/S0960129505005074.

64 Daniele Varacca and Nobuko Yoshida. Probabilistic π-Calculus and Event Structures. *Electronic Notes in Theoretical Computer Science*, 190(3):147–166, 2007. Proceedings of the Fifth Workshop on Quantitative Aspects of Programming Languages (QAPL 2007). doi:10.1016/j.entcs.2007.07.009.

65 Glynn Winskel. Probabilistic and Quantum Event Structures. In Franck van Breugel, Elham Kashefi, Catuscia Palamidessi, and Jan Rutten, editors, *Horizons of the Mind. A Tribute to Prakash Panangaden: Essays Dedicated to Prakash Panangaden on the Occasion*

of His 60th Birthday, pages 476–497, Cham, 2014. Springer International Publishing. doi:10.1007/978-3-319-06880-0_25.

66 Noam Zilberstein, Derek Dreyer, and Alexandra Silva. Outcome Logic: A Unifying Foundation for Correctness and Incorrectness Reasoning. *Proc. ACM Program. Lang.*, 7(OOPSLA1), April 2023. doi:10.1145/3586045.

67 Noam Zilberstein, Daniele Gorla, and Alexandra Silva. Denotational Semantics for Probabilistic and Concurrent Programs (Full Version), 2025. arXiv:2503.02768.

68 Noam Zilberstein, Dexter Kozen, Alexandra Silva, and Joseph Tassarotti. A demonic outcome logic for randomized nondeterminism. *Proc. ACM Program. Lang.*, 9(POPL), January 2025. doi:10.1145/3704855.

69 Noam Zilberstein, Angelina Saliling, and Alexandra Silva. Outcome separation logic: Local reasoning for correctness and incorrectness with computational effects. *Proc. ACM Program. Lang.*, 8(OOPSLA1), April 2024. doi:10.1145/3649821.

70 Noam Zilberstein, Alexandra Silva, and Joseph Tassarotti. Probabilistic Concurrent Reasoning in Outcome Logic: Independence, Conditioning, and Invariants, 2024. doi:10.48550/arXiv.2411.11662.

71 Maaike Zwart. *On the Non-Compositionality of Monads via Distributive Laws.* PhD thesis, University of Oxford, 2020. URL: https://ora.ox.ac.uk/objects/uuid:b2222b14-3895-4c87-91f4-13a8d046febb.

72 Maaike Zwart and Dan Marsden. No-Go Theorems for Distributive Laws. In *2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–13, 2019. doi:10.1109/lics.2019.8785707.

A Preliminaries and Omitted Definitions

A.1 Order Theory and Domain Theory

▶ **Definition A.1** (Poset). *A partially ordered set, or* poset*, is a set equipped with a partial order: that is, a pair $\langle X, \leq\rangle$ consisting of a set X and an order relation $\leq \subseteq X \times X$ which is reflexive, transitive and antisymmetric. When the order is instead irreflexive (*i.e., *$x \not< x$ for all $x \in X$), the poset is called* strict *and the order relation is denoted by $<$.*

Given a strict poset $\langle X, <\rangle$ and $x \in X$, we define the upward and downward closures:

$$x\!\uparrow \triangleq \{y \in X \mid x < y\} \qquad\qquad x\!\downarrow \triangleq \{y \in X \mid y < x\};$$

and its set of immediate successors and predecessors as

$$\mathsf{succ}(x) \triangleq \{y \in x\!\uparrow \mid \nexists z.(x < z < y)\} \qquad\qquad \mathsf{pred}(x) \triangleq \{y \in x\!\downarrow \mid \nexists z.(y < z < x)\}$$

All the above operations can easily be extended to sets of elements, *e.g.*, $X\!\uparrow \triangleq \bigcup_{x \in X} x\!\uparrow$, for any set X. The set of minimal and maximal nodes of a poset is given by:

$$\max \triangleq \{x \in X \mid \mathsf{succ}(x) = \emptyset\} \qquad \min \triangleq \{x \in X \mid \mathsf{pred}(x) = \emptyset\}$$

To define our new structures we need a few more definitions on posets that will play a role in proving the existence of fixed points, essential for giving a semantics to unbounded loops. First, we recall the notion of downward closed sets.

▶ **Definition A.2** (Downward Closure). *Given a poset $\langle X, <\rangle$, a set $Y \subseteq X$ is downward closed, written $Y \subseteq_\downarrow X$, iff for all $y \in Y$ it holds that $y\!\downarrow \subseteq Y$.*

Second, we need a notion of a poset being finitely preceded.

▶ **Definition A.3** (Finitely Preceded). *A poset $\langle X, <\rangle$ is* finitely preceded *if there are finitely many elements smaller than each element; that is: $|x\!\downarrow| < \infty$, for all $x \in X$.*

Third, we define the level of elements in the poset.

▶ **Definition A.4** (Level [37]). *Given a finitely preceded poset* $\langle X, < \rangle$, *define* $\mathsf{lev}\colon X \to \mathbb{N}$*:*

$$\mathsf{lev}(x) \triangleq \sup \left\{ n \;\middle|\; \exists x_0, \ldots, x_n \in X.\ x_0 \in \min \ \wedge\ x_n = x \ \wedge\ \forall i < n.\ x_{i+1} \in \mathsf{succ}(x_i) \right\}$$

We also define its inverse $\mathsf{lev}^{-1}\colon \mathbb{N} \to \mathcal{P}(X)$*:*

$$\mathsf{lev}^{-1}(n) \triangleq \{x \in X \mid \mathsf{lev}(x) = n\}$$

▶ **Definition A.5.** *The satisfaction relation* $\models\, \subseteq (\mathsf{nodes} \to \mathbb{B}) \times \mathsf{form}$ *for Boolean formulae is given below:*

$v \models \mathsf{true}$	*always*
$v \models \mathsf{false}$	*never*
$v \models \psi_1 \wedge \psi_2$	*iff* $v \models \psi_1$ *and* $v \models \psi_2$
$v \models \psi_1 \vee \psi_2$	*iff* $v \models \psi_1$ *or* $v \models \psi_2$
$v \models \neg\psi$	*iff* $v \not\models \psi$
$v \models x$	*iff* $v(x) = 1$

A.2 The Convex Powerdomain

We begin by providing the omitted definitions from Section 2, regarding the DCPO structure of the convex powerset. We start by defining the order on probability distributions. For any set X, let $X_\bot = X \cup \{\bot\}$ and the order over distributions $\sqsubseteq_{\mathcal{D}}\, \subseteq \mathcal{D}(X_\bot) \times \mathcal{D}(X_\bot)$ be defined as follows:

$$\mu \sqsubseteq_{\mathcal{D}} \nu \quad \text{iff} \quad \forall x \in X.\ \mu(x) \le \nu(x)$$

Note that this means that $\mu(\bot) \ge \nu(\bot)$ if $\mu \sqsubseteq_{\mathcal{D}} \nu$, and that the bottom of the order is $\bot_{\mathcal{D}} = [\bot \mapsto 1]$. A set of distributions $S \subseteq \mathcal{D}(X_\bot)$ is *upward closed* if, for all $\mu \in S$, it holds that $\mu \sqsubseteq_{\mathcal{D}} \nu$ implies $\nu \in S$. In addition, S is *Cauchy closed* if it is closed in the product of Euclidean topologies [36, Definition 5.4.3]. Now, the convex powerset is defined as follows:

$$\mathcal{C}(X) \triangleq \{S \subseteq \mathcal{D}(X_\bot) \;\mid\; S \text{ is nonempty, convex, upward closed, and Cauchy closed}\}$$

Convex powersets are ordered via the Smyth order [53] $\sqsubseteq_{\mathcal{C}}\, \subseteq \mathcal{C}(X) \times \mathcal{C}(X)$, defined below:

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

Since the sets above are upward closed, the Smyth order collapses to reverse subset inclusion $S \sqsubseteq_{\mathcal{C}} T$ iff $S \supseteq T$. This makes the *convex powerdomain* $\langle \mathcal{C}(X), \sqsubseteq_{\mathcal{C}} \rangle$ a pointed DCPO with suprema given by set intersection and bottom $\bot_{\mathcal{C}} = \mathcal{D}(X_\bot)$ being the set of all distributions, for the full proof refer to [68].

Finally, we give the definitions for the monad operations [23]: unit $\eta\colon X \to \mathcal{C}(X)$ and Kleisli extension $(-)^\dagger\colon (X \to \mathcal{C}(Y)) \to \mathcal{C}(X) \to \mathcal{C}(Y)$.

$$\eta(x) \triangleq [x \mapsto 1] \qquad f^\dagger(S) \triangleq \left\{ \sum_{x \in \mathsf{supp}(\mu)} \mu(x) \cdot \nu_x \;\middle|\; \mu \in S, \forall x \in \mathsf{supp}(\mu).\ \nu_x \in f_\bot(x) \right\}$$

where $(-)_\bot\colon (X \to \mathcal{C}(Y)) \to X_\bot \to \mathcal{C}(Y)$ is defined as follows: $f_\bot(x) \triangleq f(x)$ if $x \in X$ and $f_\bot(\bot) \triangleq \bot_{\mathcal{C}}$ otherwise.

$$
\begin{aligned}
[\![\textsf{skip}]\!]_{\mathcal{C}}(s) &\triangleq \eta(s)\\
[\![C_1; C_2]\!]_{\mathcal{C}}(s) &\triangleq [\![C_2]\!]_{\mathcal{C}}^{\dagger}\left([\![C_1]\!]_{\mathcal{C}}(s)\right)\\
[\![\textsf{if } b \textsf{ then } C_1 \textsf{ else } C_2]\!]_{\mathcal{C}}(s) &\triangleq \begin{cases} [\![C_1]\!]_{\mathcal{C}}(s) & \text{if } [\![b]\!]_{\mathit{test}}(s) = 1\\ [\![C_2]\!]_{\mathcal{C}}(s) & \text{if } [\![b]\!]_{\mathit{test}}(s) = 0\end{cases}\\
[\![\textsf{while } b \textsf{ do } C]\!]_{\mathcal{C}}(s) &\triangleq \mathsf{lfp}(\Psi_{\langle [\![C]\!]_{\mathcal{C}}, b\rangle})(s)\\
[\![a]\!]_{\mathcal{C}}(s) &\triangleq [\![a]\!]_{\mathit{act}}(s)
\end{aligned}
$$

Figure 4 Convex powerset semantics $[\![-]\!]_{\mathcal{C}} : \mathit{cmd} \to \mathcal{S} \to \mathcal{C}(\mathcal{S})$ due to [20, 36, 68], where Ψ is defined in Lemma 5.2.

$$
\begin{aligned}
[\![\textsf{skip}]\!]_{\mathsf{PL}} &\triangleq \{\langle \mathsf{fork}\rangle\}\\
[\![a]\!]_{\mathsf{PL}} &\triangleq \{\langle a\rangle\}\\
[\![C_1; C_2]\!]_{\mathsf{PL}} &\triangleq [\![C_1]\!]_{\mathsf{PL}} \fatsemi [\![C_2]\!]_{\mathsf{PL}}\\
[\![C_1 \mid C_2]\!]_{\mathsf{PL}} &\triangleq [\![C_1]\!]_{\mathsf{PL}} \parallel [\![C_2]\!]_{\mathsf{PL}}\\
[\![\textsf{if } b \textsf{ then } C_1 \textsf{ else } C_2]\!]_{\mathsf{PL}} &\triangleq (\{\langle \textsf{assume } b\rangle\} \fatsemi [\![C_1]\!]_{\mathsf{PL}}) \cup (\{\langle \textsf{assume } \neg b\rangle\} \fatsemi [\![C_2]\!]_{\mathsf{PL}})\\
[\![\textsf{while } b \textsf{ do } C]\!]_{\mathsf{PL}} &\triangleq \mathsf{lfp}\left(\Xi_{\langle C, b\rangle}\right)
\end{aligned}
$$

$$
\begin{aligned}
\text{where}\quad \Xi_{\langle C,b\rangle}(S) &\triangleq (\{\langle \textsf{assume } b\rangle\} \fatsemi [\![C]\!]_{\mathsf{PL}} \fatsemi S) \cup (\{\langle \textsf{assume } \neg b\rangle\} \fatsemi [\![\textsf{skip}]\!]_{\mathsf{PL}})\\
S \fatsemi T &\triangleq \{\boldsymbol{\alpha} \fatsemi \boldsymbol{\beta} \mid \boldsymbol{\alpha} \in S, \boldsymbol{\beta} \in T\}\\
S \parallel T &\triangleq \{\boldsymbol{\alpha} \parallel \boldsymbol{\beta} \mid \boldsymbol{\alpha} \in S, \boldsymbol{\beta} \in T\}
\end{aligned}
$$

Figure 5 Standard pomset language semantics $[\![-]\!]_{\mathsf{PL}} : \mathit{cmd} \to \mathit{pom}\mathcal{L}\mathit{ang}$.

As is required in Section 5.2, it is proven in [68] that the Kleisli extension for $\mathcal{C}$ defined above is Scott continuous and additive:

$$
\sup_{f \in D, x \in D'} f^{\dagger}(S) = (\sup D)^{\dagger}(\sup D') \qquad\qquad f^{\dagger}(S \,\&\, T) = f^{\dagger}(S) \,\&\, f^{\dagger}(T)
$$

Based on these operations, Figure 4 shows the convex powerset semantics for the parallel-free fragment of the language in (1), $[\![-]\!]_{\mathcal{C}} : \mathit{cmd} \to \mathcal{S} \to \mathcal{C}(\mathcal{S})$. This is precisely the same semantics used in prior work for reasoning about programs that are both nondeterministic and probabilistic [20, 36, 68]. This semantics does not capture parallel computation, which requires the more sophisticated semantic domains that we develop in this paper. The Scott continuity property above ensures that $\Psi_{\langle [\![C]\!]_{\mathcal{C}}, b\rangle} : (\mathcal{S} \to \mathcal{C}(\mathcal{S})) \to \mathcal{S} \to \mathcal{C}(\mathcal{S})$ is Scott continuous, and therefore the semantics of loops is well defined.

A.3 Pomset Languages

As we saw in Section 6, a pomset language is a set of finite pomsets without any tests or formulae. Control flow is determined by **assume** actions, which keep or eliminate traces depending on the whether the corresponding test passes or fails.

$$
\mathit{pom}\mathcal{L}\mathit{ang} \triangleq \mathcal{P}(\mathit{pom}_{\mathsf{fin}}(\mathit{label}_{\mathsf{PL}}))
$$

We give the pomset language semantics for a nondeterministic language in Figure 5.