A Denotational Semantics for Nondeterminism in Probabilistic Programs

Di Wang
Carnegie Mellon University
diw3@cs.cmu.edu

Jan Hoffmann
Carnegie Mellon University
jhoffmann@cs.cmu.edu

Thomas Reps
University of Wisconsin
GrammaTech, Inc.
reps@cs.wisc.edu

Abstract

Probabilistic programming is an increasingly popular technique for modeling randomness and uncertainty. Designing semantic models for probabilistic programs is technically challenging and has been extensively studied. A particular complication is to precisely account for nondeterminism, which is often used to represent adversarial actions in probabilistic models, and to power refinement-based development. This paper studies a denotational semantics for probabilistic programs that is based on a novel treatment of the nondeterminism, which involves nondeterminacy among *transformers* instead of *states*. The studied language combines nondeterminism with interesting features such as continuous sampling, conditioning, unstructured control-flow, general recursion, and local variables. The semantics is based on a domain-theoretic characterization of *sub-probability kernels*, defining a notion of *transition maps*, as well as constructing *powerdomains* over *transition maps* to model nondeterminism. Semantic objects in the powerdomain enjoy *general convexity*, which is a generalization of convexity. As an application, the paper studies the semantic foundations of an algebraic framework for static analysis of probabilistic programs and demonstrates that the denotational semantics is instrumental for the effectiveness of the analysis.

Keywords Probabilistic program, Denotational semantics, Powerdomain, Nondeterminism

1 Introduction

Probabilistic programming provides a powerful framework for implementing randomized algorithms [2], cryptography protocols [3], cognitive models [19], and machine learning algorithms [18]. One important focus of recent studies on probabilistic programming is to reason *rigorously* about probabilistic programs and systems (e.g. [21, 24, 30, 31, 35]). The first step in such works is to provide a suitable formal semantics for probabilistic programs (e.g., [5, 7, 14, 22, 23, 27, 28, 37–39]).

Nondeterminism is an important feature of probabilistic programming from two perspectives. First, it arises naturally from probabilistic models, such as the policy for a *Markov decision process* [4] or the unknown input distribution for modeling *fault tolerance* [25]. Second, it is required by the common paradigm of *abstraction* and *refinement*[1] on programs [13, 31]: a nondeterministic choice not only can (i) abstract deterministic conditional choices (N1), e.g., $z > 1$, but can also (ii) abstract probabilistic choices (N2), e.g., $\mathsf{Bernoulli}(0.5)$, where $\mathsf{Bernoulli}(p)$ represents a coin flip that returns *true* with probability p. While there are domain-theoretic studies that focus on nondeterminism in the sense of (N2) [12, 30, 32, 33, 40], nondeterminism in the sense of (N1) has not received much attention. One reason is of the following commonplace principle of semantics research:

> A nondeterministic function can be represented as a deterministic set-valued function, where the set contains all the values that the nondeterministic function can output for a given input.

This approach of "resolving nondeterminism last" causes (N1) to become a special case of (N2). For example, consider the program **if** $\star$ **then** A **else** B **fi**, where $\star$ represents a nondeterministic choice. We can view $\mathsf{Bernoulli}(0)$ and $\mathsf{Bernoulli}(1)$ as two candidates for the refinement of $\star$, which correspond to *false* and *true*, respectively. Abstracting from the details of the probabilistic semantics, this roughly implies that the behavior of the program contains the set $\{A, B\}$. On the other hand, if $\star$ is refined as a standard conditional choice then the refined program performs either A or B, which is included in $\{A, B\}$.

However, "resolving nondeterminism last" is not always satisfactory because it is often desirable to resolve nondeterminism *first*. For example, suppose that a program using nondeterminism is proposed as a specification of a system that is intended to be free of *side-channel-attack* vulnerabilities. Then it is desirable to show that for every implementation of the specification (i.e., a refined program with all nondeterminism resolved), its behaviors on all inputs are indistinguishable (e.g., the running times are equal). Intuitively, if a deterministic program is a function in $X \to X$ then the nondeterminism-last approach defines semantics with the signature $X \to P(X)$, while nondeterminism-first leads to $P(X \to X)$, where $P(S)$ is a collection of subsets of S. As a consequence, the nondeterminism-first resolution makes (N1) and (N2) substantially different—(N1) can observe program states while (N2) can not.

This paper develops a denotational semantics for a first-order probabilistic programming language, with nondeterminism-first resolution. The language supports the mixture of deterministic-conditional and probabilistic choices, unstructured control-flow, general recursion, local variables, as well as standard probabilistic constructs like continuous sampling and conditioning. Probabilistic programs are defined using control-flow graphs, or more precisely, *hyper-graphs*, which are capable of describing unstructured control-flow in probabilistic programming. The denotational semantics of a probabilistic program is defined directly with respect to its control-flow hyper-graph, as a least solution to the equation system transformed from the hyper-graph. General recursion and local variables are added to the language in a standard way.

For each probabilistic program, the semantics defines a *semantic object*, which is a subset with some desirable properties, and each element of the subset represents the meaning of a refined version

[1] Abstraction enables reasoning about a program through its high-level specifications, and refinement allows stepwise software development, where programs are "refined" from specifications to low-level implementations.

PL'17, January 01–03, 2017, New York, NY, USA
2017.

of the program (i.e, all nondeterministic choices are resolved as deterministic ones). Technically, this means that the semantic domain is a convex powerdomain over sub-probability kernels. However, there is a gap between measure-theoretic studies on kernels and domain-theoretic studies on powerdomains. The mixture of two nondeterminism also requires a generalized form of convexity. This paper defines a notion of *transition maps*, a domain-theoretic characterization of kernels, as well as develops Hoare and Smyth powerdomains over transition maps, corresponding to partial and total correctness of programs, respectively. This paper also defines a notion of *general convexity* of semantic objects, describing each object is stable under arbitrary (possibly mixed) deterministic choices—as the standard convexity is used to describe stability under arbitrary probabilistic choices.

We also illustrate an application of the denotational semantics—to prove the soundness of a static analysis framework of probabilistic programs [41]. The properties of our powerdomain constructions motivate an algebraic design of abstract domains used for static analyses. The soundness for the analysis framework is proved with respect to the denotational semantics, based on a notion of probabilistic abstractions.

2 Background

In this section, we first introduce probabilistic programming by example and then discuss existing semantic models for probabilistic programming languages.

2.1 Probabilistic Programming

We briefly discuss several important and interesting features of probabilistic programming, based on some examples of arithmetic probabilistic programs with real-valued variables.

Discrete and continuous sampling. Fig. 1a illustrates a mixed model of Gaussian distributions. Bernoulli(0.5) is a *sampling expression* that represents a fair coin flip. Sampling expressions can be treated as inhabitants of the boolean type. As a result, we can mix standard and probabilistic choices, e.g., Bernoulli(0.5) $\wedge\, n \geq 10$. The collection of such mixed choices is called *deterministic choices*. The *sampling statement* $x \sim$ Gaussian(0, 1) draws a value from a Gaussian distribution, and assigns it to the variable x. In Fig. 1a, based on outcome of the coin flip, x is sampled from a Gaussian distribution with mean 0 or mean 1, and standard derivation 1.

Conditioning. Fig. 1b draws two independent values from the uniform distribution on the interval $(0, 1)$, assigns them to x and y respectively, and conditions possible program states on the observation $x > y$. Intuitively, the program expresses priori knowledge about x and y and then a measurement determines that x is greater than y. The probability distribution at the end of Fig. 1b should put weight 0.5 on those combinations of x and y such that $x > y$, and weight 0 on other combinations.

```
(a)
if Bernoulli(0.5) then
  x ~ Gaussian(0, 1)
else
  x ~ Gaussian(1, 1)
fi

(b)
x ~ Uniform(0, 1);
y ~ Uniform(0, 1);
observe(x > y)

(c)
z ~ Uniform(0, 2);
if ★ then x := x + z
else y := y + z
fi
```

Figure 1. (a) Sampling; (b) Conditioning; (c) Nondeterminism

```
(a)
n := 0;
while Bernoulli(0.9) do
  n := n + 1;
  if n ≥ 10 then break
  else continue
od

(b)
proc p begin
  if Bernoulli(0.6) then skip
  else
    t := t + 1; call p;
    t := t + 1; call p
  fi
end
```

Figure 2. (a) Unstructured control-flow; (b) Recursion

Nondeterminism. The program in Fig. 1c samples z from a uniform distribution on the interval $(0, 2)$ and then *nondeterministically* adds it to either x or y. The symbol $\star$ stands for a *nondeterministic choice* that can behave like arbitrary deterministic choices. Intuitively, a nondeterministic program is executed with respect to an *oracle*, which interprets every nondeterministic choice in the execution as a deterministic one. In Fig. 1c, an oracle can interpret $\star$ as a standard choice $z > 1$, or as a probabilistic choice Bernoulli(0.5), or even as a mixed choice $z > 1 \wedge$ Bernoulli(0.5).

Unstructured control-flow. As in standard programs, unstructured control-flow is introduced by **goto**, **break**, and/or **continue** statements. Fig. 2a presents a probabilistic while-loop which models a variant geometric distribution, where all probabilities of $n > 10$ accumulate to the probability of $n = 10$. The **break** statement exits the while-loop, and the **continue** statement jumps to the loop head.

General recursion. Fig. 2b shows an example of recursive probabilistic programs. It defines a procedure p, in which there are two procedure calls **call** p to itself. When a call to the procedure p returns, the increase in the variable t records the number of procedure calls during the execution. Note that our language also permits mutual recursion, i.e., two procedures can call each other.

2.2 Semantic Models

Semantics for probabilistic programming languages have been extensively studied.

Deterministic models. We first discuss semantics for probabilistic programming languages without nondeterminism. Kozen [27] provides a classic semantics for probabilistic programs in terms of distribution transformers. To reduce redundancies, other modern approaches use probability kernels [28, 37], sub-probability kernels [7], and s-finite kernels [5, 38]. A different approach uses measurable functions $A \longrightarrow P(\mathbb{R}_{\geq 0} \times B)$ where $P(S)$ stands for the set of all probability measures on S [39]. For higher-order languages, Jones and Plotkin [22, 23] have developed a probabilistic powerdomain that is a set of continuous evaluations on a state space. They show that the powerdomain can be used to solve recursive domain equations. Ehrhard et al. [14] provide a Cartesian-closed category on stable and measurable maps between cones, and use it to give a semantics for probabilistic PCF.

As baseline and starting point for our development, we adopt Borgström et al.'s approach [7] to introduce an operational model for a simple probabilistic language without nondeterminism in §3.

Nondeterministic models. Many works use weakest pre-expectations to reason about probabilistic programs with nondeterminism. McIver and Morgan [30, 31] develop probabilistic predicate transformers. Jansen et al. [21] extend the reasoning to support conditioning. Kaminski et al. [24] develop a weakest-precondition logic for expected run-times. Olmedo et al. further adopt the reasoning to recursive programs, for analysis of both predicates and expected run-times [35]. They use an operational semantics that is based on Markov decision procedures.

Other studies on combining probability and nondeterminism utilize domain-theoretic powerdomain constructions. Informally, powerdomain means "nondeterministic construct", and the semantics in these studies has the form $X \to \mathcal{P}(\mathcal{D}(X))$, where X is the state space, $\mathcal{D}(X)$ denotes the set of distributions over X, and $\mathcal{P}(S)$ is a powerdomain over S. McIver and Morgan build a Plotkin-style powerdomain over probability distributions on a discrete state space [30]. Mislove et al. [32, 33] study powerdomain constructions for probabilistic CSP. Tix et al. [40] generalize McIver and Morgan's results to continuous state spaces, and construct three powerdomains for the extended probabilistic powerdomain.

As discussed in §1, we are interested in developing a semantics that resolves nondeterminism *first*, and *then* input. Intuitively, the form of our semantics should be $\mathcal{P}(X \to \mathcal{D}(X))$.

3 A Measure-Theoretic Operational Semantics

In this section, we sketch a measure-theoretic operational semantics for an imperative, single-procedure, and deterministic probabilistic programming language, following the approach of Borgström et al.'s distribution-based semantics [7]. We use the operational semantics to illustrate (i) how to use probability distributions and kernels in the semantics, and (ii) how to model executions of probabilistic programs operationally.

3.1 A Hyper-Graph Model of Probabilistic Programs

We define the operational semantics on *control-flow graphs* of programs. We adopt a common approach for standard control-flow graphs, in which the nodes represent program locations, and edges labeled with instructions describe transitions among program locations (e.g., [15, 29, 34]). Instead of standard directed graphs, we make use of *hyper-graphs* [17].

Definition 3.1 (Hyper-graphs)**.** A *hyper-graph* H is a quadruple $\langle V, E, v^{\mathsf{entry}}, v^{\mathsf{exit}} \rangle$, where V is a finite set of nodes, E is a set of hyper-edges, $v^{\mathsf{entry}} \in V$ is a distinguished *entry node*, and $v^{\mathsf{exit}} \in V$ is a distinguished *exit node*. A *hyper-edge* is an ordered pair $\langle x, Y \rangle$, where $x \in V$ is a node and $Y \subseteq V$ is an ordered, non-empty set of nodes. For a hyper-edge $e = \langle x, Y \rangle$ in E, we use $src(e)$ to denote x and $Dst(e)$ to denote Y. Following the terminology from graphs, we say that e is an *outgoing edge* of x and an *incoming edge* of each of the nodes $y \in Y$. We assume v^{entry} has no incoming edges, and v^{exit} has no outgoing edges.

Definition 3.2 (Probabilistic programs)**.** A *probabilistic program* contains a finite set of procedures $\{\langle H_i, LV_i \rangle\}_{1 \le i \le n}$, where each procedure is a pair where $H_i = \langle V_i, E_i, v_i^{\mathsf{entry}}, v_i^{\mathsf{exit}} \rangle$ is a control-flow hyper-graph in which each node except v_i^{exit} has *at least* one outgoing hyper-edge, v_i^{exit} has no outgoing edge, and LV_i is a finite set of local variables. We assume that the nodes and local variables of each procedure are pairwise disjoint. To assign meanings to probabilistic programs modulo *data actions* $\mathcal{A}$ and *deterministic conditions* $\mathcal{L}$, we associate with each hyper-edge $e \in E = \bigcup_{1 \le i \le n} E_i$ a

$\mathcal{A} ::= x := e \mid x \sim \mathcal{D} \mid \textbf{observe}(\varphi) \mid \textbf{skip}$
$\varphi \in \mathcal{L} ::= \textbf{true} \mid \textbf{false} \mid e \bowtie u$ where $\bowtie \in \{=, \le, \ge\} \mid \neg\varphi \mid \mathcal{BD}$
$e, u \in \mathsf{Exp} ::= x \mid c$ where $c \in \mathbb{R} \mid e \bullet u$ where $\bullet \in \{+, -, \times, \div\}$
$x \in \mathsf{Var} \supseteq \bigcup_{1 \le i \le n} LV_i ::= x \mid y \mid z \mid \cdots$
$\mathcal{BD} \in \mathsf{BDist} ::= \mathsf{Bernoulli}(e) \mid \mathsf{sampleBool}_f(\overline{e})$
$\mathcal{D} \in \mathsf{Dist} ::= \mathsf{Uniform}(e, u) \mid \mathsf{Gaussian}(e, u) \mid \mathsf{sampleReal}_f(\overline{e})$

Figure 3. Examples of data actions and deterministic conditions

control-flow action $Ctrl(e)$, where $Ctrl$ is

$$\begin{array}{rcl} Ctrl & ::= & seq[\mathsf{act}] \text{ where } \mathsf{act} \in \mathcal{A} \mid cond[\varphi] \text{ where } \varphi \in \mathcal{L} \\ & \mid & call[i \to j] \text{ where } 1 \le i, j \le n \end{array}$$

where the number of destination nodes $|Dst(e)|$ of a hyper-edge e is 1 if $Ctrl(e)$ is $seq[\mathsf{act}]$ or $call[i \to j]$, and 2 otherwise.

See Fig. 3 for data actions $\mathcal{A}$ and deterministic conditions $\mathcal{L}$ that would be used for an arithmetic program. $\mathsf{sampleBool}_f(\overline{e})$ samples a Boolean value with respect to the probability density function f, where $\overline{e}$ is a vector of parameters, while $\mathsf{sampleReal}_f(\overline{e})$ samples a real value. For example, $\mathsf{Gaussian}(e, u)$ can be represented as $\mathsf{sampleReal}_f(e, u)$ where $f = \lambda(\mu, \sigma).\lambda x. \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$. Note that **goto**, **break**, and **continue** are not data actions, and are encoded directly as edges in control-flow hyper-graphs in a standard way.

Example 3.3. Fig. 4 shows the control-flow hyper-graph of the program in Fig. 2a, where v_0 is the entry and v_4 is the exit. The hyper-edge $\langle v_2, \{v_3\} \rangle$ is associated with a sequencing action $seq[n := n + 1]$, while $\langle v_1, \{v_2, v_4\} \rangle$ is assigned a deterministic-choice action $cond[\mathsf{Bernoulli}(0.9)]$.

3.2 Background from Measure Theory

To define semantics for probabilistic programs modulo data actions $\mathcal{A}$ and deterministic conditions $\mathcal{L}$, we review some standard definitions from measure theory [6, 36].

A *measurable space* is a pair $\langle X, \Sigma \rangle$ where X is a non-empty set called the *sample space*, and Σ is a σ-algebra over X (i.e., a set of subsets of X which contains $\emptyset$ and is closed under complement and countable union). A *measurable function* from a measurable space $\langle X_1, \Sigma_1 \rangle$ to a measurable space $\langle X_2, \Sigma_2 \rangle$ is a mapping $f : X_1 \to X_2$ such that for all $A \in \Sigma_2$, $f^{-1}(A) \in \Sigma_1$. The measurable functions from a measurable space $\langle X, \Sigma \rangle$ to a Borel space $\mathcal{B}(\mathbb{R}_{\ge 0})$ on nonnegative real numbers (the smallest σ-algebra containing all open intervals) are called Σ-measurable.

A *measure* μ on a measurable space $\langle X, \Sigma \rangle$ is a function from Σ to $[0, \infty]$ such that: (i) $\mu(\emptyset) = 0$, and (ii) for all pairwise-disjoint countable sequences of sets $A_1, A_2, \cdots \in \Sigma$ (i.e., $A_i \cap A_j = \emptyset$ for all $i \ne j$) we have $\sum_{i=1}^{\infty} \mu(A_i) = \mu(\bigcup_{i=1}^{\infty} A_i)$. The measure μ is called a *(sub-probability) distribution* if $\mu(X) \le 1$. If μ is a distribution and $c \in [0, 1]$, we write $c \cdot \mu$ for the distribution $\lambda A.c \cdot \mu(A)$. If μ, ν are distributions and $c, d \in [0, 1]$ such that $c + d \le 1$, we write $c \cdot \mu + d \cdot \nu$ for the distribution $\lambda A.c \cdot \mu(A) + d \cdot \nu(A)$. A *measure space* is a triple

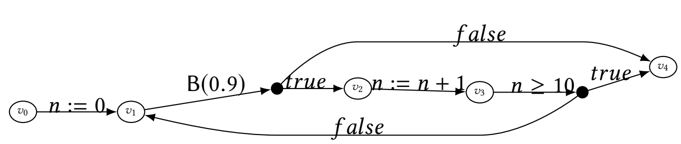

Figure 4. Control-flow hyper-graph of the program in Fig. 2a

$$\begin{array}{ll}
\widehat{x := e} = \lambda\omega.\delta((x \mapsto e(\omega))\omega) & \widehat{\mathbf{true}} = \lambda\omega.1 \\
\widehat{x \sim \mathscr{D}} = \lambda\omega.\lambda F.\mu_{\mathscr{D}}(\{v \mid (x \mapsto v)\omega \in F\}) & \widehat{\mathbf{false}} = \lambda\omega.0 \\
\widehat{\mathbf{observe}}(\varphi) = \lambda\omega.\widehat{\varphi}(\omega) \cdot \delta(\omega) & \widehat{e \bowtie u} = \lambda\omega.[e(\omega) \bowtie u(\omega)] \\
\widehat{\mathbf{skip}} = \lambda\omega.\delta(\omega) & \widehat{\neg\varphi} = \lambda\omega.1 - \widehat{\varphi}(\omega) \\
 & \widehat{\mathscr{BD}} = \lambda\omega.p_{\mathscr{BD}}(true)
\end{array}$$

Figure 5. Interpretation of actions and conditions

$\mathscr{M} = \langle X, \Sigma, \mu \rangle$ where μ is a measure on the measurable space $\langle X, \Sigma \rangle$. The integral of a Σ-measurable function f over the measure space $\mathscr{M} = \langle X, \Sigma, \mu \rangle$ can be defined following Lebesgue's theory and denoted either by $\int f d\mu$ or $\int f(x)\mu(dx)$. The *Dirac measure* $\delta(x)$ is defined as $\lambda A.[x \in A]$, where $[\varphi]$ is an *Iverson bracket*, which evaluates to 1 if φ is true, and 0 otherwise. The *zero measure* $\mathbf{0}$ is defined as $\lambda A.0$.

A *(sub-probability) kernel* from a measurable space $\langle X_1, \Sigma_1 \rangle$ to a measurable space $\langle X_2, \Sigma_2 \rangle$ is a function $\kappa : X_1 \to \Sigma_2 \to [0, 1]$ such that:[2] (i) for each x in X_1, the function $\kappa(x)$ is a distribution on $\langle X_2, \Sigma_2 \rangle$, and (ii) for each A in Σ_2, the function $\lambda x.\kappa(x)(A)$ is Σ_1-measurable. We write the integral of a Σ_2-measurable function f with respect to the distribution $\kappa(x)$ in (i) as $\int f(y)\kappa(x)(dy)$. The property (ii) is used to define kernel composition.

Intuitively, kernels describe transformers from state space X_1 to X_2.[3] Let us consider kernels over a single state space X. A sequence of actions $\text{act}_1; \text{act}_2$ with κ_1 and κ_2, respectively, is modeled by their *composition*, defined as follows:[4]

$$\lambda x.\lambda A. \int \kappa_1(x)(dy)\kappa_2(y)(A). \tag{1}$$

3.3 An Operational Semantics

The next step is to define semantics based on the control-flow hyper-graphs. We adopt Borgström et al.'s distribution-based small-step operational semantics for lambda calculus [7] to our imperative program model, while we suppress the features of multi-procedure and nondeterminism for now.

Three components are used to define the semantics:

- A *measurable space* $\mathscr{P} = \langle \Omega, \mathcal{F} \rangle$ over program states (e.g., finite mappings from program variables to values).
- A mapping from data actions act to *kernels* $\widehat{\text{act}} : \Omega \to \mathcal{F} \to [0, 1]$ from Ω to itself. The intuition to keep in mind is that $\widehat{\text{act}}(\omega)(F)$ is the probability that the action, starting in state $\omega \in \Omega$, halts in a state that satisfies $F \in \mathcal{F}$ [28].
- A mapping from deterministic conditions φ to measurable functions $\widehat{\varphi} : \Omega \to [0, 1]$.

For an arithmetic program with a finite set Var of program variables, Ω is defined as $\text{Var} \to \mathbb{R}$ and the measurable state space $\mathscr{P}$ is defined as the Borel space on Ω. Fig. 5 shows interpretation of the data actions and deterministic conditions in Fig. 3, where $e(\omega)$ evaluates expression e in state ω, $(x \mapsto v)\omega$ updates x in ω with v, $\mu_{\mathscr{D}} : \mathcal{B}(\mathbb{R}) \to [0, 1]$ is the measure corresponding to the distribution $\mathscr{D}$ on reals, and $p_{\mathscr{BD}} : \{true, false\} \to [0, 1]$ is the probability mass function corresponding to the distribution $\mathscr{BD}$ on Booleans.

[2] Kernels are originally formalized as $X_1 \times \Sigma_2 \to [0, 1]$. We use the curried version.
[3] As explained by Kozen [28], for finite or countable X_1, X_2, a kernel has a representation as a Markov transition matrix M, in which each entry $M(x_1, x_2)$ for a pair of states $\langle x_1, x_2 \rangle$ gives the probability that x_1 transitions to x_2.
[4] For finite or countable state spaces X, and the matrix representation described in footnote 3, the integral in Eqn. (1) degenerates to matrix multiplication [28].

Suppose that $P = \langle V, E, v^{\text{entry}}, v^{\text{exit}} \rangle$ is a single-procedure deterministic program. Therefore, each node in P except v^{exit} is associated with *exactly* one hyper-edge. The *program configurations* $T = V \times \Omega$ are pairs of the form $\langle v, \omega \rangle$, where $v \in V$ is a node in the control-flow hyper-graph, and $\omega \in \Omega$ is a program state. Because V is a finite set of nodes and $\mathscr{V} \stackrel{\text{def}}{=} \langle V, 2^V \rangle$ is naturally a measurable space, we define $\mathscr{T}$ as the product space of $\mathscr{V}$ and $\mathscr{P}$ to be a measurable space over program configurations T.

We then define *one-step evaluation* as a relation $\langle v, \omega \rangle \longrightarrow \mu$ between configurations $\langle v, \omega \rangle$ and distributions μ on configurations, as shown in Fig. 6.

$$\begin{array}{lll}
\langle v, \omega \rangle & \longrightarrow & \lambda A.\widehat{\text{act}}(\omega)(\{\omega' \mid \langle u, \omega' \rangle \in A\}) \\
 & \text{where} & e = \langle v, \{u\} \rangle \in E, Ctrl(e) = seq[\text{act}] \\
\langle v, \omega \rangle & \longrightarrow & \widehat{\varphi}(\omega) \cdot \delta(\langle u_1, \omega \rangle) + (1 - \widehat{\varphi}(\omega)) \cdot \delta(\langle u_2, \omega \rangle) \\
 & \text{where} & e = \langle v, \{u_1, u_2\} \rangle \in E, Ctrl(e) = cond[\varphi]
\end{array}$$

Figure 6. One-step evaluation relation

Example 3.4. In Fig. 4, some one-step evaluations are

$$\begin{array}{l}
\langle v_0, n \mapsto 233 \rangle \longrightarrow \lambda A.[\langle v_1, n \mapsto 0 \rangle \in A] \\
\langle v_1, n \mapsto 1 \rangle \longrightarrow \lambda A.0.9 \cdot [\langle v_2, n \mapsto 1 \rangle \in A] + 0.1 \cdot [\langle v_4, n \mapsto 1 \rangle \in A] \\
\langle v_3, n \mapsto 9 \rangle \longrightarrow \lambda A.[\langle v_1, n \mapsto 9 \rangle \in A]
\end{array}$$

Lemma 3.5. *$\longrightarrow$ is a kernel.*

Then we define *step-indexed evaluation* as the family of n-indexed relations $\langle v, \omega \rangle \longrightarrow_n \mu$ between configurations $\langle v, \omega \rangle$ and distributions μ on program states inductively, as shown in Fig. 7.

$$\begin{array}{rll}
\langle v, \omega \rangle & \longrightarrow_0 & \mathbf{0} \\
\langle v^{\text{exit}}, \omega \rangle & \longrightarrow_n & \delta(\omega) \quad \text{if } n > 0 \\
\langle v, \omega \rangle & \longrightarrow_{n+1} & \lambda F. \int \mu_\tau(F)\mu(d\tau) \\
 & \text{where} & \langle v, \omega \rangle \longrightarrow \mu \\
 & \text{and} & \tau \longrightarrow_n \mu_\tau \text{ for any } \tau \in supp(\mu)
\end{array}$$

Figure 7. Step-indexed evaluation relation

Example 3.6. In Fig. 4, some step-indexed evaluations are

$$\begin{array}{l}
\langle v_4, n \mapsto 10 \rangle \longrightarrow_1 \lambda F.[(n \mapsto 10) \in F] \\
\langle v_1, n \mapsto 0 \rangle \longrightarrow_2 \lambda F.0.1 \cdot [(n \mapsto 0) \in F] \\
\langle v_1, n \mapsto 0 \rangle \longrightarrow_5 \lambda F.0.1 \cdot [(n \mapsto 0) \in F] + 0.09 \cdot [(n \mapsto 1) \in F]
\end{array}$$

Lemma 3.7. *$\longrightarrow_n$ is a kernel for all n.*

Because the set of distributions with the pointwise order forms an ω-cpo (i.e., an ω-complete partial order), we define the semantics of the program $P = \langle V, E, v^{\text{entry}}, v^{\text{exit}} \rangle$ as

$$[\![P]\!]_{\text{os}} \stackrel{\text{def}}{=} \lambda\omega. \sup_{n \in \mathbb{N}} \{\mu \mid \langle v^{\text{entry}}, \omega \rangle \longrightarrow_n \mu\}.$$

Example 3.8. In Fig. 4, $[\![P]\!]_{\text{os}}(\omega)$ for any initial state ω is given by

$$\lambda F. \sum_{k=0}^{9} (0.1 \times 0.9^k) \cdot [(n \mapsto k) \in F] + 0.3486784401 \cdot [(n \mapsto 10) \in F].$$

Theorem 3.9. *$[\![P]\!]_{\text{os}}$ is a kernel.*

3.4 Adding Nondeterminism

For probabilistic programming, nondeterminism is often introduced by the notion of a *scheduler*, which resolves a nondeterministic choice from the computation that leads up to it (e.g., [8, 9, 16]).

When the scheduler is fixed, a program can be interpreted deterministically. In this paper, a nondeterministic choice abstracts every possibly probabilistic deterministic choice (e.g., $z > 1 \wedge \mathsf{Bernoulli}(0.5)$), which only observe *current* program states. As a consequence, we consider *memoryless* schedulers, which make decisions based on current program states.

To introduce nondeterminism to the hyper-graph model, we allow some nodes to have *two* outgoing hyper-edges. We then annotate a one-step evaluation with the edge e on which it evaluates, as $\longrightarrow^e$. Let $\mathscr{P} = \langle \Omega, \mathcal{F} \rangle$ be the measurable space over program states. We define a *scheduler* σ as a measurable function $\Omega \to [0,1]$. A *scheduler trace* t is a finite sequence $[\sigma_1, \cdots, \sigma_m]$ of schedulers. We redefine step-indexed evaluations as $\xrightarrow{t}_n$, where t is a scheduler trace, in Fig. 8.

$$
\begin{array}{rcll}
\langle v, \omega \rangle & \xrightarrow{t}_0 & \mathbf{0} & \\
\langle v^{\mathsf{exit}}, \omega \rangle & \xrightarrow{t}_n & \delta(\omega) \quad \text{if } n > 0 & \\
\langle v, \omega \rangle & \xrightarrow{t}_{n+1} & \lambda F. \int \mu_\tau(F) \mu(d\tau) & \\
& \text{where} & v \text{ has 1 outgoing edge } e, \langle v, \omega \rangle \longrightarrow^e \mu & \\
& \text{and} & \tau \xrightarrow{t}_n \mu_\tau \text{ for any } \tau \in supp(\mu) & \\
\langle v, \omega \rangle & \xrightarrow{\sigma :: t}_{n+1} & \sigma(\omega) \cdot \mu_1 + (1 - \sigma(\omega)) \cdot \mu_2 & \\
& \text{where} & v \text{ has 2 outgoing edges } e_1, e_2 & \\
& \text{and} & \langle v, \omega \rangle \longrightarrow^{e_i} \nu_i, \mu_i = \lambda F. \int \mu_\tau(F) \nu_i(d\tau) & \\
& & \tau \xrightarrow{t}_n \mu_\tau \text{ for any } \tau \in supp(\nu_i) &
\end{array}
$$

Figure 8. Annotated step-indexed evaluation relation

We define the set of all scheduler traces as $\mathbb{T} = \bigcup_{m \in \mathbb{N}} \mathbb{T}_m$, where $\mathbb{T}_m$ is the set of all scheduler traces with length m. For each m, we collect a set S_m of kernels:

$$S_m = \{\lambda\omega. \sup_{n \in \mathbb{N}} \{\mu \mid \langle v^{\mathsf{entry}}, \omega \rangle \xrightarrow{t}_n \mu\} \mid t \in \mathbb{T}_m\}$$

Intuitively, S_m describes all program refinements in which no more than m nondeterministic choices are resolved. Then we want to define the nondeterministic semantics as the "limit" of the sequence $\{S_m\}_{m \in \mathbb{N}}$, i.e., $\lim_{m \to \infty} S_m$.

However, it is unclear whether the "limit" exists or not. We need to study the structure of S_m and characterize its properties. Observing that S_m is an element of the *power set* of kernels, we turn to utilize existing studies of *powerdomains*. Unfortunately, domain-theoretic studies on powerdomains require the underlying set to be a continuous (or even coherent) domain, which is fundamentally different from the measurable spaces with which kernels are defined. We address this challenge in §4 and §5.

On the other hand, for rigorous reasoning about program meanings, it is more desirable to develop the semantics in a *compositional* manner—that is, the property of a whole program can be established from properties of its proper constituents. The nondeterministic operational semantics proposed above is not compositional—it always looks at the whole program. Therefore, we develop a *denotational* semantics in §6.

4 Domain-Theoretic Characterization of Kernels

In this section, we first review some standard notions from domain theory [1]. We develop domain-theoretic characterizations of kernels. We study general convexity, setting the stage for development of powerdomain constructions over transition maps in §5.

4.1 Background from Domain Theory

Let X be a partially ordered set, i.e., a *poset*. A subset of D of X is *directed* if it is nonempty and each pair of elements of D has an upper bound in D. If a directed set D has a supremum, then it is denoted by $\bigvee D$. X is called *directed complete* or a *dcpo* if each directed subset D has a supremum $\bigvee D$ in X. A dcpo X is *pointed* if X contains a least element. A function $f : X \to Y$ between two dcpos is *Scott-continuous* if it is monotone and preserves directed joins, i.e., $f(\bigvee D) = \bigvee f(D)$ for all directed subsets D. The set of all Scott-continuous functions between X and Y is denoted as $[X \to Y]$.

The *lower closure* of a subset A is defined as $\downarrow A \stackrel{\text{def}}{=} \{x \in X \mid \exists a \in A. x \le a\}$. The *upper closure* of a subset A is defined as $\uparrow A \stackrel{\text{def}}{=} \{x \in X \mid \exists a \in A. a \le x\}$. A subset A with $\downarrow A = A$ is called a *lower set*. A subset A with $\uparrow A = A$ is called an *upper set*. A subset A of a dcpo X is *Scott-closed* if A is a lower set and if $\bigvee D \in A$ for every directed set $D \subseteq A$. The complement $X \setminus A$ of a Scott-closed set A is called *Scott-open*. The Scott-open sets form a topology on X. For any topological space X we denote the collection of open sets by $\mathcal{O}(X)$. A function $f : X \to Y$ between two topological spaces is *topologically continuous* if for all $U \in \mathcal{O}(Y)$, $f^{-1}(U) \in \mathcal{O}(X)$. Between dcpos, topologically continuous functions are precisely the Scott-continuous functions with respect to Scott topologies [1, Prop. 2.3.4]. In the rest of this paper, we sometimes call these *continuous* functions.

Let X be a dcpo. For two elements x, y of X, we say that x *approximates* y, denoted by $x \ll y$, if for all directed subsets D of X, $y \le \bigvee D$ implies $x \le d$ for some $d \in D$. We define $\Downarrow A \stackrel{\text{def}}{=} \{x \in X \mid \exists a \in A. x \ll a\}$ and $\Uparrow A \stackrel{\text{def}}{=} \{x \in X \mid \exists a \in A. a \ll x\}$. X is called *continuous* if for all x of X, the set $\Downarrow x$ is directed and $x = \bigvee \Downarrow x$. A continuous dcpo is also called a *continuous domain*.

The *closure* of a subset A of a dcpo X is the smallest Scott-closed set containing A, denoted by $\overline{A}$. A *cover* C of a subset A of a dcpo X is a collection of subsets of X whose union contains A as a subset. A *sub-cover* of C is a subset of C that still covers A. C is called an *open-cover* if each of its member is an open set. A subset A of a dcpo X is *compact* if every open-cover of A contains a finite sub-cover. A subset A of a dcpo X is *saturated* if A is intersection of its neighborhoods. The saturation of compact sets is also compact. With respect to the Scott topology, saturated sets are precisely the upper sets.

4.2 Transition Maps

We now review a domain-theoretic characterization of distributions (e.g., as in [23, 40]), and extend the ideas to kernels.

Let X be a dcpo. A function $\mu : \mathcal{O}(X) \to [0,1]$ is called a *valuation* on X if: (i) $\mu(\emptyset) = 0$, (ii) $U \subseteq V$ implies $\mu(U) \le \mu(V)$ for all $U, V \in \mathcal{O}(X)$, and (iii) $\mu(U) + \mu(V) = \mu(U \cup V) + \mu(U \cap V)$ for all $U, V \in \mathcal{O}(X)$. If μ is also Scott-continuous, i.e., $\mu(\bigcup_{i \in I} U_i) = \bigvee_{i \in I} \mu(U_i)$ for all directed collections of open sets in $\mathcal{O}(X)$, then μ is called a *continuous valuation*. The set of all continuous valuations on X is denoted by $\mathcal{D}(X)$. Given that X is a dcpo, valuations are defined with respect to the Scott topology, and they are ordered pointwise.

The integration of a continuous function f from X to $[0,1]$ with respect to a valuation μ is defined following Lebesgue's theory, denoted as $\int f d\mu$.

Now we define the notion of transition maps. Let X be a dcpo. A continuous function $\kappa \in [X \to \mathcal{D}(X)]$ is called a *transition map*. Let $\mathcal{K}(X) \stackrel{\text{def}}{=} [X \to \mathcal{D}(X)]$. Then $\mathcal{K}(X)$ is naturally a dcpo with the pointwise order, denoted by $\sqsubseteq$. We will show the definition of transition maps is reasonable with respect to that of kernels.

Lemma 4.1. *Let κ be a continuous function in $[X \to \mathcal{D}(X)]$.*
- *For all x of X, $\kappa(x)$ is a valuation.*
- *For all U of $\mathcal{O}(X)$, $\lambda x.\kappa(x)(U)$ is a continuous function.*

We write the integral of a continuous function f with respect to the valuation obtained by giving an element x to a transition map κ as $\int f(y)\kappa(x)(dy)$. We then define *composition* of transition maps κ_1, κ_2 as $\kappa_1 \otimes \kappa_2 \stackrel{\text{def}}{=} \lambda x.\lambda U. \int \kappa_1(x)(dy)\kappa_2(y)(U)$, and *conditional-choice* of transition maps κ_1, κ_2 conditioning on f as $\kappa_1 \,{}_f\!\diamond \kappa_2 \stackrel{\text{def}}{=} \lambda x. f(x) \cdot \kappa_1(x) + (1 - f(x)) \cdot \kappa_2(x)$, where f is a continuous function from X to $[0, 1]$.

Example 4.2. Recall the development of an operational semantics in §3.3. We can now reformulate it in a domain-theoretic way.
- Ω is a dcpo over program states, with a Scott topology. For arithmetic programs, the state space Ω is defined as $\mathrm{Var} \to (\mathbb{R} \cup \{+\infty\})$, with the pointwise order.
- Data actions are interpreted as *transition maps* $\widehat{\mathrm{act}} \in \mathcal{K}(\Omega)$. For simplicity, suppose the program only has one variable and Ω is treated as $\mathbb{R} \cup \{+\infty\}$. The open sets in $\mathcal{O}(\Omega)$ have the form $(a, +\infty]$ for any $a \in \mathbb{R} \cup \{+\infty\}$. The Dirac measure $\delta(x)$ is represented as $\lambda(a, +\infty].[x > a]$ in $\mathcal{D}(\Omega)$. The distribution with probability density function f can be denoted as $\lambda(a, +\infty]. \int_{x>a} f(x)dx$, which is also a continuous valuation. Using the methodology of these two constructs, we can reformulate interpretations of data actions in Fig. 5 as transition maps.
- We interpret deterministic conditions as *continuous* function from Ω to $[0, 1]$. The interpretation in Fig. 5 is still valid.

The rest of §3.3 also holds for transition maps—for a probabilistic program P, the operational semantics $[\![P]\!]_{\mathrm{os}}$ for it can be derived as a transition map. We will override $[\![P]\!]_{\mathrm{os}}$ to denote the domain-theoretic operational semantics in the following sections.

Here we state a useful lemma:

Lemma 4.3. *$\otimes$ and ${}_f\!\diamond$ are Scott-continuous.*

4.3 General Convexity

For standard program semantics, a nondeterministic choice of two semantic objects is usually interpreted as *set union*, i.e., the result can exhibit behaviors from both objects. In probabilistic programming, a nondeterministic choice can also model a probabilistic one. For example, if semantic object A establishes property φ and B establishes ψ, then a nondeterministic choice of A and B should include a semantic object that establishes φ with probability 0.3 and ψ with 0.7. Therefore, if the underlying set of the semantic domain is equipped with addition and scalar multiplication, the nondeterministic choice of A and B should at least model the meaning of $\{p \cdot a + (1-p) \cdot b \mid a \in A \wedge b \in B \wedge 0 \le p \le 1\}$. This set is the *convex combination* of A and B. On the other hand, the nondeterministic choice of A and A is usually supposed to be A exactly. As a consequence, every semantic object should be closed under convex combination.

However, a more complicated notion of complexity is needed to develop semantics over transition maps. Let X be the state space. Every semantic object should be closed under the ${}_f\!\diamond$ operator for every feasible function f from X to $[0, 1]$. Recall that the definition $\kappa_1 \,{}_f\!\diamond \kappa_2 = \lambda x. f(x) \cdot \kappa_1(x) + (1 - f(x)) \cdot \kappa_2(x)$ is similar to a convex combination, except the coefficients can not only be constants, but depend on the state $x \in X$. We formalize the idea by defining a notion of *general convexity*.

Let X be a dcpo. A subset S of $\mathcal{K}(X)$ is called *generally convex* if for all $\kappa_1, \kappa_2 \in S$ and all continuous functions f from X to $[0, 1]$, the conditional-choice $\kappa_1 \,{}_f\!\diamond \kappa_2$ is contained in S. General convexity is a generalization of standard convexity, which is defined with respect to constant functions.

We show that some operations preserve general convexity.

Lemma 4.4. *Let S be a generally convex subset of $\mathcal{K}(X)$. Then*
- *The closure $\overline{S}$ is generally convex.*
- *The saturation $\uparrow S$ and the lower closure $\downarrow S$ are generally convex.*

Lemma 4.5. *Suppose S_1 and S_2 are generally convex subsets of $\mathcal{K}(X)$. Then $\{a \,{}_f\!\diamond b \mid a \in S_1 \wedge b \in S_2\}$ is generally convex for all continuous function f from X to $[0, 1]$.*

The *generally convex hull* of a subset A of $\mathcal{K}(X)$ is the smallest generally convex set containing A, denoted by *conv*(A).

Example 4.6. Suppose $X = \mathbb{R} \cup \{+\infty\}$. Let $\kappa_1 \stackrel{\text{def}}{=} \lambda x.\lambda(a, +\infty].[x + 1 > a]$ and $\kappa_2 \stackrel{\text{def}}{=} \lambda x.\lambda(a, +\infty].[x - 1 > a]$. Intuitively, κ_1 and κ_2 describe the actions $x := x+1$ and $x := x-1$, respectively. By definition of general convexity, the singleton sets $\{\kappa_1\}$ and $\{\kappa_2\}$ are generally convex. Then the general convex hull of $\{\kappa_1, \kappa_2\}$ should contain all conditional-choice of two transition maps. *conv*$(\{\kappa_1, \kappa_2\})$ can be given by $\{\lambda x.\lambda(a, +\infty]. f(x) \cdot [x + 1 > a] + (1 - f(x)) \cdot [x - 1 > a] \mid f \in [X \to [0, 1]]\}$. Intuitively, the nondeterministic choice **if** $\star$ **then** $x := x + 1$ **else** $x := x - 1$ **fi** should at least model the meaning of *conv*$(\{\kappa_1, \kappa_2\})$.

Following are some properties of the *conv*$(\cdot)$ operator.

Lemma 4.7. *If S_1 and S_2 are generally convex, then conv$(S_1 \cup S_2)$ is given by* $\{\kappa_1 \,{}_f\!\diamond \kappa_2 \mid \kappa_1 \in S_1 \wedge \kappa_2 \in S_2 \wedge f \text{ continuous}\}$.

Lemma 4.8. *Let S_1 and S_2 be compact generally convex subsets of $\mathcal{K}(X)$. Then conv$(S_1 \cup S_2)$ is also compact.*

For a finite set F, by a simple induction we have *conv*$(F) = \{\lambda x. \sum_{\kappa \in F} f_\kappa(x) \cdot \kappa(x) \mid f_\kappa \in [X \to [0, 1]], \sum f_\kappa = \dot{1}\}$.

Lemma 4.9. *For an arbitrary $S \subseteq \mathcal{K}(X)$, we have*

$$conv(S) = \bigcup_{F \subseteq S,\, F \text{ finite}} conv(F).$$

5 Powerdomains over Transition Maps

In this section, we first review some results from domain-theoretic studies on powerdomains [1]. We then develop two powerdomain constructions over transition maps, corresponding to *partial correctness* and *total correctness*, respectively.

5.1 A Sketch of Powertheories

The *Plotkin powertheory* is defined by one binary operation $\Cup$ and the following laws: (i) $x \Cup y = y \Cup x$ for all x, y, (ii) $(x \Cup y) \Cup z = x \Cup (y \Cup z)$ for all x, y, z, and (iii) $x \Cup x = x$ for all x. The operation

$\Cup$ is called *formal union*. The *Hoare powertheory* is the Plotkin powertheory augmented by the inequality $x \sqsubseteq x \Cup y$, where $\sqsubseteq$ is a partial order. Similarly, the *Smyth powertheory* is the Plotkin powertheory augmented by the inequality $x \sqsupseteq x \Cup y$.

There is an interesting connection between powerdomain constructions and *partial/total correctness* used in program verification [20]. Intuitively, the formal union $\Cup$ is interpreted as nondeterministic-choice, but partial and total correctness treat it from different viewpoints. Partial correctness requires a program behaves correctly if it terminates. Total correctness additionally requires the program *does* terminate. Therefore, if a program A satisfies φ under partial correctness, then $A \Cup B$ should also satisfy φ for any B, i.e., the nondeterministic-choice behaves *angelically*. On the other hand, if a program $A \Cup B$ satisfies φ under total correctness, then both A and B must satisfy φ, i.e., the nondeterministic-choice behaves *demonically*. If we interpret the partial order $\sqsubseteq$ as an *abstraction* order—$x \sqsubseteq y$ means x is an abstraction of y—if x satisfies some property, then so does y. Then we can establish the following connections:

- $x \sqsubseteq x \Cup y$ coincides with partial correctness, hence the Hoare construction stands for partial correctness.
- $x \sqsupseteq x \Cup y$ coincides with total correctness, hence the Smyth construction stands for total correctness.

Example 5.1. Consider the following three programs:

$$x \sim \mathsf{Uniform}(0,1) \tag{2}$$

$$\textbf{if } \star \textbf{ then } x \sim \mathsf{Uniform}(0,1) \textbf{ else observe}(\textbf{false}) \textbf{ fi} \tag{3}$$

$$\textbf{observe}(\textbf{false}) \tag{4}$$

Assume we model observation failure as nontermination. When reasoning about partial correctness, the programs (2) and (3) are indifferent, because partial correctness does not distinguish termination and nontermination. It indicates that the Hoare semantics should give the same semantics to (2, 3). On the other hand, the program (3) and (4) are indistinguishable in total correctness, because they do not always terminate. It indicates that the Smyth semantics should give the same semantics to (3, 4).

As a consequence, the Plotkin construction is capable of describing both partial and total correctness. In this paper, we succeed in developing Hoare and Smyth powerdomains over transition maps—but not in constructing the Plotkin one yet. To see the reason, we review some representation theorems about powerdomains.

A continuous domain X is *coherent* if the intersection of two compact saturated sets is again compact.

Two subsets A and B of a set equipped with a binary relation R are in the *Egli-Milner relation*, written as $A \mathrel{\mathsf{R}_{EM}} B$, if the following conditions hold: (i) for all a of A, there exists b in B such that $a \mathrel{\mathsf{R}} b$, and (ii) for all b of B, there exists a in A such that $a \mathrel{\mathsf{R}} b$.

A *lens* on a dcpo X is a subset of X that is the intersection of a Scott-closed subset and a Scott-compact saturated subset.

Theorem 5.2. *The following representation theorems hold:*

- *The Hoare powerdomain of a dcpo X is isomorphic to the lattice of all nonempty Scott-closed subsets of X [1, Thm. 6.2.13].*
- *The Smyth powerdomain of a continuous domain X is isomorphic to the collection of nonempty Scott-compact saturated subsets of X ordered by reversed inclusion [1, Thm. 6.2.14].*
- *The Plotkin powerdomain of a coherent domain X is isomorphic to the collections of lenses of X ordered by the Egli-Milner order [1, Thm. 6.2.22].*

Without any constraints on X, we can only prove that $\mathcal{K}(X)$ is a dcpo. We provide a solution to adding constraints to X in order to make $\mathcal{K}(X)$ a continuous domain, but leave the question how to make $\mathcal{K}(X)$ coherent for future work.

5.2 An Axiomatic Characterization of Powerdomains

We now motivate our constructions by an axiomatic characterization of desirable properties. Let X be a dcpo. We want to construct a powerdomain over $\mathcal{K}(X)$, which is a subset of $2^{\mathcal{K}(X)}$. We denote an instance of such a powerdomain as $\mathbb{P}X$, Then we want to lift composition $\otimes$ and conditional-choice ${}_f\diamond$ to the powerdomain, denoted as $\otimes_{\mathbb{P}}$ and ${}_f\diamond_{\mathbb{P}}$, respectively.

Nondeterministic-choice. Intuitively, nondeterministic-choice should be idempotent, commutative, and associative. The following holds in the traditional semantics.

- For a program A, a nondeterministic choice between A and A itself should behave exactly the same as A.
- For two programs A and B, a nondeterministic choice between A and B should be the same of the choice between B and A.
- For three programs A, B, and C, first deciding if executing A, and then deciding between B and C if A is not chosen, should be indifferent from first deciding if executing C, and then deciding between A and B, if C is not chosen, because exactly one program among A, B, C is to be executed in either case.

In other words, the nondeterministic-choice is essentially a formal-union operator $\Cup_{\mathbb{P}}$ on $\mathbb{P}X$.

Composition. For two elements A, B of $\mathbb{P}X$, $A \otimes_{\mathbb{P}} B$ should contain all pairwise compositions of transition maps in A and B, i.e., $\{\kappa_1 \otimes \kappa_2 \mid \kappa_1 \in A \wedge \kappa_2 \in B\}$. Furthermore, there should be an identity element $\underline{1}_{\mathbb{P}}$ in $\mathbb{P}X$ representing the semantics of **skip** statements, i.e., $A \otimes_{\mathbb{P}} \underline{1}_{\mathbb{P}} = \underline{1}_{\mathbb{P}} \otimes_{\mathbb{P}} A = A$ for all A in $\mathbb{P}X$. $\otimes_{\mathbb{P}}$ also needs to be associative, because programs should compose. In other words, $\otimes_{\mathbb{P}}$ is a *monoid* operator on $\mathbb{P}X$.

Conditional-choice. For two elements A, B of $\mathbb{P}X$, and a continuous function f from X to $[0,1]$, $A\ {}_f\diamond_{\mathbb{P}} B$ should contain all pairwise conditional-choices of transition maps in A and B, i.e., $\{\kappa_1\ {}_f\diamond \kappa_2 \mid \kappa_1 \in A \wedge \kappa_2 \in B\}$. Furthermore, some variants of idempotence, commutativity, and associativity listed in Fig. 9 should hold. These laws are desirable because we consider conditional-choice as a deterministic choice that depends on current program state.

$$\boxed{\begin{gathered} A\ {}_f\diamond_{\mathbb{P}} A = A \\ A\ {}_f\diamond_{\mathbb{P}} B = B\ {}_{\dot{1}-f}\diamond_{\mathbb{P}} A \\ (A\ {}_f\diamond_{\mathbb{P}} B)\ {}_g\diamond_{\mathbb{P}} C = A\ {}_{f'}\diamond_{\mathbb{P}} (B\ {}_{g'}\diamond_{\mathbb{P}} C) \\ \text{where } f' = f \cdot g \text{ and } (\dot{1} - f') \cdot (\dot{1} - g') = \dot{1} - g \end{gathered}}$$

Figure 9. Laws for conditional-choice

5.3 The Hoare Construction

Let X be a dcpo. We consider the collection

$$\mathbb{H}X \stackrel{\text{def}}{=} \{S \subseteq \mathcal{K}(X) \mid S \text{ nonempty, generally convex, Scott-closed}\}$$

ordered by $A \sqsubseteq_{\mathbb{H}} B \stackrel{\text{def}}{=} A \subseteq\downarrow B$, which is equivalent to inclusion $\subseteq$.

Lemma 5.3. $\langle \mathbb{H}X, \subseteq \rangle$ *is a dcpo.*

The bottom element in $\mathbb{H}X$ is defined as $\underline{\perp}_{\mathbb{H}} \stackrel{\text{def}}{=} \{\lambda x.\lambda U.0\}$. The directed suprema in $\mathbb{H}X$ is performed as $\bigvee A_i = \overline{\bigcup A_i}$. Then we can lift kernel composition $\otimes$ and conditional-choice ${}_f\diamond$ to the powerdomain $\mathbb{H}X$.

$$A \otimes_{\mathbb{H}} B \stackrel{\text{def}}{=} \overline{conv\{a \otimes b \mid a \in A \wedge b \in B\}}$$
$$A\,{}_f\diamond_{\mathbb{H}} B \stackrel{\text{def}}{=} \overline{\{a\,{}_f\diamond b \mid a \in A \wedge b \in B\}}$$

Lemma 5.4. *$\otimes_{\mathbb{H}}$ and ${}_f\diamond_{\mathbb{H}}$ are Scott-continuous, $\langle \mathbb{H}X, \otimes_{\mathbb{H}}, \underline{1}_{\mathbb{H}}\rangle$ is a monoid where $\underline{1}_{\mathbb{H}} \stackrel{def}{=}\downarrow \{\lambda x.\lambda U.[x \in U]\}$, and laws in Fig. 9 hold for ${}_f\diamond_{\mathbb{H}}$.*

The formal union operation in $\mathbb{H}X$ is defined as $A \Cup_{\mathbb{H}} B \stackrel{\text{def}}{=} \overline{conv(A \cup B)}$.

Lemma 5.5. *$\Cup_{\mathbb{H}}$ is Scott-continuous, idempotent, commutative, and associative.*

Example 5.6. Consider the three programs shown in Ex. 5.1. Suppose $X = \mathbb{R} \cup \{+\infty\}$. The transition maps for $x \sim \mathsf{Uniform}(0,1)$ and **observe**(**false**) are $\kappa_1 = \lambda x.\lambda(a, +\infty]. \max(0, \min(1, 1-a))$ and $\kappa_2 = \lambda x.\lambda(a, +\infty].0$, respectively. Then program (2) and (4) are represented as $\downarrow \{\kappa_1\}$ and $\{\kappa_2\}$ in $\mathbb{H}X$, respectively. Note that $\{\kappa_2\}$ is actually $\underline{\perp}_{\mathbb{H}}$. Because $\kappa_2 \sqsubseteq \kappa_1$, we know that $\{\kappa_2\} \subseteq\downarrow \{\kappa_1\}$. The semantics of program (3) is then derived as $\downarrow \{\kappa_1\} \Cup_{\mathbb{H}} \{\kappa_2\} = \overline{conv(\downarrow \{\kappa_1\} \cup \{\kappa_2\})} =\downarrow \{\kappa_1\}$, which is the same as the semantics of program (2).

5.4 The Smyth Construction

Let X be a dcpo. However, $\mathcal{K}(X) = [X \to \mathcal{D}(X)]$ is not always continuous. We review the notion of *FS-domains* to resolve the issue.

Let X be a dcpo and $f : X \to X$ be a continuous function. f is *finitely separate* from the identity on X if there exists a finite set M such that for any x of X there is $m \in M$ with $f(x) \le m \le x$. A pointed dcpo X is called an *FS-domain* if there is a directed collection $\{f_i\}_{i \in I}$ of continuous functions on X, each finitely separated from identity, with the identity map as their supremum.

Theorem 5.7. *If X is an FS-domain and Y is pointed and continuous, then $[X \to Y]$ is continuous [1, Prop. 4.2.10].*

Let X be an FS-domain. Then X is continuous and hence $\mathcal{D}(X)$ is pointed and continuous [40, Thm. 2.10]. Therefore $\mathcal{K}(X) = [X \to \mathcal{D}(X)]$ is continuous by Thm. 5.7.

We consider the collection

$$\mathbb{S}X \stackrel{\text{def}}{=} \{S \subseteq \mathcal{K}(X) \mid S \text{ nonempty, generally convex, Scott-compact, saturated}\}$$

ordered by $A \sqsubseteq_{\mathbb{S}} B \stackrel{\text{def}}{=}\uparrow A \supseteq B$, which is equivalent to reverse inclusion $\supseteq$.

Lemma 5.8. *$\langle \mathbb{S}X, \supseteq\rangle$ is a dcpo.*

The bottom element in $\mathbb{S}X$ is defined $\underline{\perp}_{\mathbb{S}} \stackrel{\text{def}}{=} \mathcal{K}(X)$. The directed suprema in $\mathbb{S}X$ is performed as $\bigvee A_i = \bigcap A_i$. Then we can lift kernel composition $\otimes$ and conditional-choice ${}_f\diamond$ to the powerdomain $\mathbb{S}X$.

$$A \otimes_{\mathbb{S}} B \stackrel{\text{def}}{=} \uparrow conv\{a \otimes b \mid a \in A \wedge b \in B\}$$
$$A\,{}_f\diamond_{\mathbb{S}} B \stackrel{\text{def}}{=} \uparrow \{a\,{}_f\diamond b \mid a \in A \wedge b \in B\}$$

Lemma 5.9. *$\otimes_{\mathbb{S}}$ and ${}_f\diamond_{\mathbb{S}}$ are Scott-continuous, $\langle \mathbb{S}X, \otimes_{\mathbb{S}}, \underline{1}_{\mathbb{S}}\rangle$ is a monoid where $\underline{1}_{\mathbb{S}} \stackrel{def}{=} \{\lambda x.\lambda U.[x \in U]\}$, and the laws in Fig. 9 hold for ${}_f\diamond_{\mathbb{S}}$.*

The formal union operation in $\mathbb{S}X$ is defined as $A \Cup_{\mathbb{S}} B \stackrel{\text{def}}{=}\uparrow conv(A \cup B)$.

Lemma 5.10. *$\Cup_{\mathbb{S}}$ is Scott-continuous, idempotent, commutative, and associative.*

Example 5.11. Let $X = \mathbb{R} \cup \{+\infty\}$, which is an FS-domain. Consider the three programs in Ex. 5.1 and recall the discussion in Ex. 5.6. The program (2) and (4) are represented as $\{\kappa_1\}$ and $\uparrow \{\kappa_2\}$ in $\mathbb{S}X$, respectively. Note that $\uparrow \{\kappa_2\}$ is actually $\underline{\perp}_{\mathbb{S}}$. Because $\kappa_2 \sqsubseteq \kappa_1$, we know that $\uparrow \{\kappa_2\} \supseteq \{\kappa_1\}$. The semantics of program (3) is then derived as $\{\kappa_1\} \Cup_{\mathbb{S}} \uparrow \{\kappa_2\} =\uparrow conv(\{\kappa_1\} \cup \uparrow \{\kappa_2\}) =\uparrow \{\kappa_2\}$, which is the same as the semantics of program (4).

6 A Domain-Theoretic Denotational Semantics

The operational semantics described in §3.3 and later reformulated in Ex. 4.2 presents a reasonable model for evaluating probabilistic programs without nondeterminism. However, a denotational semantics is more suitable to reason about program properties, because it abstracts away how a program is evaluated and concentrates only on the effect of the program.

In this section, we first develop a denotational semantics for the restricted programming language in §3.3 and show its equivalence to the domain-theoretic operational semantics. We then consider features like nondeterminism, recursion, and local variables.

6.1 A Denotational Semantics for a Restricted Language

For standard programs, a denotational semantics can assign to a control-flow node v either backward meanings—about the computations that can lead up to v—for forward meanings—about the computations that can continue from v [10, 11]. Because we work with hyper-graphs rather than standard directed graphs, there is a difference in how things "look" in the backward and forward direction: hyper-edges fan *out* in the forward direction. Hyper-edges can have two destination nodes, but only one source node.

When there is no nondeterminism, we can assign a single transition map to every control-flow node v, which represents the effects from v to the exit node. Recall the three components used to define semantics:

- A dcpo $\mathscr{P} = \Omega$ with a Scott topology over program states.
- A mapping from data actions act to transition maps $\widehat{\mathsf{act}} \in \mathcal{K}(\Omega)$.
- A mapping from logical conditions φ to continuous functions $\widehat{\varphi} \in [\Omega \to [0,1]]$.

Given a probabilistic program $P = \langle V, E, v^{\mathsf{entry}}, v^{\mathsf{exit}}\rangle$, let $\mathcal{S}\langle v\rangle \in \mathcal{K}(\Omega)$ be the semantics assigned to the node v; the following local properties should hold:

- if $e = \langle v, \{u_1, \cdots, u_k\}\rangle \in E$, then $\mathcal{S}\langle v\rangle = \widehat{Ctrl(e)}(\mathcal{S}\langle u_1\rangle, \cdots, \mathcal{S}\langle u_k\rangle)$, and
- otherwise, $\mathcal{S}\langle v\rangle = \lambda\omega.\lambda F.[\omega \in F]$.

The function $\widehat{Ctrl(e)}$ for different kinds of control-flow actions is defined as follows:

$\widehat{seq[\mathsf{act}]}(\kappa_1) \stackrel{\text{def}}{=} \widehat{\mathsf{act}} \otimes \kappa_1$	$\widehat{cond[\varphi]}(\kappa_1, \kappa_2) \stackrel{\text{def}}{=} \kappa_1\,{}_{\widehat{\varphi}}\diamond \kappa_2$

Example 6.1. Recall the control-flow hyper-graph in Fig. 4. It can be transformed to the equation system in Fig. 10. Then the semantics can be represented as a solution to the system.

We can then define a function F_P whose *fixed points* satisfy the local properties above:

$$\lambda\mathcal{S}.\lambda v.\begin{cases}\widehat{Ctrl(e)}(\mathcal{S}(u_1),\cdots,\mathcal{S}(u_k)) & e=\langle v,\{u_1,\cdots,u_k\}\rangle\in E\\ \lambda\omega.\lambda F.[\omega\in F] & \text{otherwise}\end{cases}$$

Recall Kleene's fixed point theorem:

Theorem 6.2. *Suppose $\langle X,\le\rangle$ is a dcpo with a least element $\bot$, and let $f : X\to X$ be a Scott-continuous function. Then f has a least fixed point, denoted by $\mathrm{lfp}^{\le}_{\bot} f$.*

By the Scott-continuity of $\otimes$ and ${}_f\diamond$ (stated in Lem. 4.3), we derive the Scott-continuity of F_P.

Lemma 6.3. *The function F_P is Scott-continuous on $\langle V\to\mathcal{K}(\Omega),\dot\sqsubseteq\rangle$, which is a dcpo with the least element $\lambda v.\underline{\bot}_{\mathbb{K}}$, where $\underline{\bot}_{\mathbb{K}}\overset{def}{=}\lambda\omega.\lambda F.0$.*

Hence we define the semantics of the program P as $[\![P]\!]_{\mathrm{ds}}\overset{\mathrm{def}}{=}(\mathrm{lfp}^{\dot\sqsubseteq}_{\lambda v.\underline{\bot}_{\mathbb{K}}}F_P)(v^{\mathrm{entry}})$. We can then prove its equivalence to the operational semantics $[\![\cdot]\!]_{\mathrm{os}}$ described in Ex. 4.2.

Theorem 6.4. $[\![P]\!]_{\mathrm{os}}=[\![P]\!]_{\mathrm{ds}}$.

6.2 Nondeterminism and Recursion

We now add nondeterminism and recursion to the restricted language in §6.1. Recall the original definition of probabilistic programs in Defn. 3.2. Nondeterminism is introduced by allowing each control-flow nodes except the exit nodes to to have *at least* one outgoing edge. Recursion is introduced by enabling multiple procedures and the *calling* actions $call[i\to j]$, which denotes a procedure call from the i-th procedure to the j-th procedure.

We denote the powerdomain construction used to define semantics by $\mathbb{P}\Omega$, which is either $\mathbb{H}\Omega$ or $\mathbb{S}\Omega$, depending on if we want to reason about partial correctness or total correctness, respectively. Given a probabilistic program $P=\{H_i\}_{1\le i\le n}$, let $\mathcal{S}\langle v\rangle\in\mathbb{P}\Omega$ be the semantics assigned to the node v; the following local properties should hold:

- if $v\ne v_i^{\mathrm{exit}}$, then $\mathcal{S}\langle v\rangle=\Cup^{\mathbb{P}}_{e=\langle v,\{u_1,\cdots,u_k\}\rangle\in E}\widehat{Ctrl(e)}(\mathcal{S}\langle u_1\rangle,\cdots,\mathcal{S}\langle u_k\rangle)$, and
- otherwise, $\mathcal{S}\langle v\rangle=\underline{1}_{\mathbb{P}}$.

The function $\widehat{Ctrl(e)}$ for different kinds of control-flow actions is defined as follows:

$\widehat{seq[\mathrm{act}]}(S_1)\overset{\mathrm{def}}{=}\mathbb{P}\langle\widehat{\mathrm{act}}\rangle\otimes_{\mathbb{P}}S_1$	$\widehat{cond[\varphi]}(S_1,S_2)\overset{\mathrm{def}}{=}S_1\ {}_{\widehat{\varphi}}\diamond_{\mathbb{P}}S_2$
$\widehat{call[i\to j]}(S_1)\overset{\mathrm{def}}{=}\mathcal{S}\langle v_j^{\mathrm{entry}}\rangle\otimes_{\mathbb{P}}S_1$	

where $\mathbb{P}\langle\kappa\rangle$ is the most precise representation of $\{\kappa\}$ in the powerdomain $\mathbb{P}\Omega$. In $\mathbb{H}\Omega$, it is the lower closure; while in $\mathbb{S}\Omega$, it is the upper closure.

$$\mathcal{S}\langle v_0\rangle=\widehat{seq[n:=0]}(\mathcal{S}\langle v_1\rangle)\qquad \mathcal{S}\langle v_3\rangle=\widehat{cond[n>10]}(\mathcal{S}\langle v_5\rangle,\mathcal{S}\langle v_1\rangle)$$
$$\mathcal{S}\langle v_1\rangle=\widehat{cond[\mathrm{B}(0.9)]}(\mathcal{S}\langle v_2\rangle,\mathcal{S}\langle v_5\rangle)\qquad \mathcal{S}\langle v_4\rangle=\lambda\omega.\lambda F.[\omega\in F]$$
$$\mathcal{S}\langle v_2\rangle=\widehat{seq[n:=n+1]}(\mathcal{S}\langle v_3\rangle)$$

Figure 10. The system corresponding to Fig. 4

Similarly, we can then define a function F_P whose fixed points satisfy the local properties above:

$$\lambda\mathcal{S}.\lambda v.\begin{cases}\Cup^{\mathbb{P}}_{e=\langle v,\{u_1,\cdots,u_k\}\rangle\in E}\widehat{Ctrl(e)}(\mathcal{S}(u_1),\cdots,\mathcal{S}(u_k)) & v\ne v_i^{\mathrm{exit}}\\ \underline{1}_{\mathbb{P}} & \text{otherwise}\end{cases}$$

Lemma 6.5. *The function F_P is Scott-continuous on $\langle V\to\mathbb{P}\Omega,\dot\sqsubseteq_{\mathbb{P}}\rangle$, which is a dcpo with the least element $\lambda v.\underline{\bot}_{\mathbb{P}}$.*

Hence we define the semantics of the procedure H_i as $[\![H_i]\!]_{\mathrm{ds}}\overset{\mathrm{def}}{=}(\mathrm{lfp}^{\dot\sqsubseteq_{\mathbb{P}}}_{\lambda v.\underline{\bot}_{\mathbb{P}}}F_P)(v_i^{\mathrm{entry}})$.

6.3 Local Variables

We can extend the semantics in §6.2 to handle local variables following a standard approach introduced by Knoop and Steffen [26]. Suppose every procedure H_i in a probabilistic program P has a finite set of local variables LV_i. At a call site where procedure H_i calls procedure H_j via a calling action $call[i\to j]$, the values of local variables in LV_i are recorded and inaccessible to H_j and procedures transitively called by H_j—we introduce a *projection* operator to restore the values of local variables after H_j returns.

Definition 6.6. The *projection* on a variable x of a semantic object A is also a semantic object, denoted by $\mathrm{Project}[x](A)$, where the following hold, for any $\bullet\in\{\otimes,{}_f\diamond,\Cup\}$:

$$\mathrm{Project}[x](\mathrm{Project}[y](A))=\mathrm{Project}[y](\mathrm{Project}[x](A))$$
$$\mathrm{Project}[x](\mathrm{Project}[x](A)\bullet_{\mathbb{P}}B)=\mathrm{Project}[x](A)\bullet_{\mathbb{P}}\mathrm{Project}[x](B)$$
$$\mathrm{Project}[x](A\bullet_{\mathbb{P}}\mathrm{Project}[x](y))=\mathrm{Project}[x](A)\bullet_{\mathbb{P}}\mathrm{Project}[x](B)$$

We also require the operator $\mathrm{Project}[x]$ to be Scott-continuous.

With the projection operator, we can interpret the calling action as $\widehat{call[i\to j]}(S_1)\overset{\mathrm{def}}{=}\mathrm{Project}[LV_i](\mathcal{S}\langle v_j^{\mathrm{entry}}\rangle)\otimes_{\mathbb{P}}S_1$. Because $\mathrm{Project}[x]$ is Scott-continuous, Lem. 6.5 still holds.

We then show a concrete example of projection operators. Suppose the language is arithmetic and all program variables are real-valued. Let Var be the set of program variables; the set of program states is $\Omega=\mathrm{Var}\to(\mathbb{R}\cup\{+\infty\})$. Then for a variable $x\in\mathrm{Var}$ and transition map κ, we define $\exists x(\kappa)\overset{\mathrm{def}}{=}\lambda\omega.\lambda F.(\kappa\otimes\widehat{x:=\omega(x)})(\omega)(F)$. For Hoare construction, we define $\mathrm{Project}_{\mathbb{H}}[x](A)\overset{\mathrm{def}}{=}\overline{\{\exists x(\kappa)\mid\kappa\in A\}}$. For Smyth construction, we define $\mathrm{Project}_{\mathbb{S}}[x](A)\overset{\mathrm{def}}{=}\uparrow\{\exists x(\kappa)\mid\kappa\in A\}$.

Lemma 6.7. *We state the continuity of projection operators.*

- $\exists x(\cdot)$ *is a Scott-continuous operator on* $\mathcal{K}(\Omega)$.
- $\mathrm{Project}_{\mathbb{H}}[x](\cdot)$ *is a Scott-continuous operator on* $\mathbb{H}\Omega$.
- $\mathrm{Project}_{\mathbb{S}}[x](\cdot)$ *is a Scott-continuous operator on* $\mathbb{S}\Omega$.

7 Application: Soundness of Static Analysis

In this section, we discuss an application of the new denotational semantics as the concrete semantics of a static analysis framework for probabilistic programs. More details about the static analysis and its soundness proof can be found in a companion paper [41].

Definition 7.1 (Pre-Markov algebras). A *pre-Markov algebra* (PMA) over a set of deterministic conditions $\mathcal{L}$ is a 7-tuple $\mathcal{M}=\langle M,\sqsubseteq,\otimes,{}_{\varphi}\diamond,\Cup,\underline{\bot},\underline{1}\rangle$, where $\langle M,\sqsubseteq,\underline{\bot}\rangle$ forms an ω-cpo with a least element $\underline{\bot}$; $\langle M,\otimes,\underline{1}\rangle$ forms a monoid (i.e., $\otimes$ is an associative binary operator with $\underline{1}$ as its identity element); ${}_{\varphi}\diamond$ is a binary operator parametrized by φ which is a condition in $\mathcal{L}$; $\Cup$ is a binary operator that is idempotent, commutative, and associative; $\otimes$, ${}_{\varphi}\diamond$, and $\Cup$ are pre-ω-continuous and the following properties hold:

$a\,{}_{\varphi}\!\diamond b \sqsubseteq a \Cup b$	$a \sqsubseteq a\,{}_{\mathbf{true}}\!\diamond b$
$a \sqsubseteq a\,{}_{\varphi}\!\diamond a$	$a\,{}_{\varphi}\!\diamond b = b\,{}_{\neg\varphi}\!\diamond a$
$(a\,{}_{\varphi}\!\diamond b)\,{}_{\psi}\!\diamond c = a\,{}_{\varphi'}\!\diamond(b\,{}_{\psi'}\!\diamond c)$	where $\varphi' = \varphi \wedge \psi, \varphi' \vee \psi' = \psi$

The precedence of the operators is that $\otimes$ binds tightest, followed by ${}_{\varphi}\diamond$ and $\Cup$.

Lemma 7.2. *The denotational semantics in §6.2 is a PMA $C = \langle \mathbb{P}\Omega, \sqsubseteq_{\mathbb{P}}, \otimes_{\mathbb{P}}, {}_{\widehat{\varphi}}\diamond_{\mathbb{P}}, \Cup_{\mathbb{P}}, \underline{\perp}_{\mathbb{P}}, \underline{1}_{\mathbb{P}} \rangle$.*

Definition 7.3 (Interpretations)**.** An interpretation is a pair $\mathscr{I} = \langle \mathcal{M}, [\![\cdot]\!] \rangle$, where $\mathcal{M}$ is a pre-Markov algebra, and $[\![\cdot]\!] : \mathcal{A} \to \mathcal{M}$, where $\mathcal{A}$ is the set of data actions for probabilistic programs. We call $\mathcal{M}$ the *semantic algebra* of the interpretation and $[\![\cdot]\!]$ the *semantic function.*

Given a probabilistic program P and an interpretation $\mathscr{I} = \langle \mathcal{M}, [\![\cdot]\!] \rangle$, We define $F_P^{\sharp}$ as

$$\lambda \mathcal{S}^{\sharp}.\lambda v. \begin{cases} \displaystyle\Cup_{e=\langle v,\{u_1,\cdots,u_k\}\rangle \in E} \widehat{Ctrl(e)}^{\sharp}(\mathcal{S}^{\sharp}(u_1), \cdots, \mathcal{S}^{\sharp}(u_k)) & v \neq v_i^{\mathsf{exit}} \\ \underline{1} & \text{otherwise} \end{cases}$$

where

$\widehat{seq[\mathsf{act}]}^{\sharp}(a_1) \overset{\text{def}}{=} [\![\mathsf{act}]\!] \otimes a_1$	$\widehat{cond[\varphi]}^{\sharp}(a_1, a_2) \overset{\text{def}}{=} a_1\,{}_{\varphi}\!\diamond a_2$
$\widehat{call[i \to j]}^{\sharp}(a_1) \overset{\text{def}}{=} \mathcal{S}^{\sharp}(v_j^{\mathsf{entry}}) \otimes a_1$	

We use the least prefixed point of $F_P^{\sharp}$ to define the interpretation of a probabilistic program P as $\mathscr{I}[P] = \mathrm{lpp}^{\dot{\sqsubseteq}}_{\lambda v.\underline{\perp}} F_P^{\sharp}$, where lpp denotes the *least prefixed point* (i.e., the least ρ such that $f(\rho) \leq \rho$ for a function f). The interpretation of a control-flow node v is then define as $\mathscr{I}[v] = \mathscr{I}[P](v)$.

Definition 7.4 (Probabilistic abstractions)**.** A *probabilistic under-abstraction*[5] from a PMA C to a PMA $\mathcal{Y}$ is a concretization mapping, $\gamma : \mathcal{Y} \to C$, such that

- γ is monotone, i.e., for all $Q_1, Q_2 \in \mathcal{Y}$, $Q_1 \sqsubseteq_{\mathcal{A}} Q_2$ implies $\gamma(Q_1) \sqsubseteq_C \gamma(Q_2)$,
- $\gamma(\underline{\perp}_{\mathcal{Y}}) \sqsubseteq_C \underline{\perp}_C$,
- $\gamma(\underline{1}_{\mathcal{Y}}) \sqsubseteq_C \underline{1}_C$,
- for all $Q_1, Q_2 \in \mathcal{Y}$, $\gamma(Q_1 \otimes_{\mathcal{Y}} Q_2) \sqsubseteq_C \gamma(Q_1) \otimes_C \gamma(Q_2)$,
- for all $Q_1, Q_2 \in \mathcal{Y}$, $\gamma(Q_1\,{}_{\varphi}\!\diamond_{\mathcal{Y}} Q_2) \sqsubseteq_C \gamma(Q_1)\,{}_{\varphi}\!\diamond_C \gamma(Q_2)$, and
- for all $Q_1, Q_2 \in \mathcal{Y}$, $\gamma(Q_1 \Cup_{\mathcal{Y}} Q_2) \sqsubseteq_C \gamma(Q_1) \Cup_C \gamma(Q_2)$.

A probabilistic abstraction leads to a sound analysis:

Theorem 7.5. *Let $\mathscr{C}$ and $\mathscr{Y}$ be interpretations over PMAs C and $\mathcal{Y}$; let γ be a probabilistic under-abstraction from C to $\mathcal{Y}$; and let P be an arbitrary program. If for all data actions* act, $\gamma([\![\mathsf{act}]\!]^{\mathscr{Y}}) \sqsubseteq_C [\![\mathsf{act}]\!]^{\mathscr{C}}$, *then for all control-flow nodes v of P, we have $\gamma(\mathscr{Y}[v]) \sqsubseteq_C \mathscr{C}[v]$.*

References

[1] S. Abramsky and A. Jung. 1994. Domain Theory. In *Handbook of Logic in Computer Science.* Oxford University Press Oxford, UK.

[2] G. Barthe, T. Espitau, M. Gaboardi, B. Grégoire, J. Hsu, and P.-Y. Strub. 2016. *Formal Certification of Randomized Algorithms.* Technical Report. http://justinh.su/files/papers/ellora.pdf.

[3] G. Barthe, B. Grégoire, and S. Zanella Béguelin. 2009. Formal Certification of Code-based Cryptographic Proofs. In *Princ. of Prog. Lang.*

[4] R. Bellman. 1957. A Markovian Decision Process. *Indiana Univ. Math. J.* (1957).

[5] B. Bichsel, T. Gehr, and M. Vechev. 2018. Fine-grained Semantics for Probabilistic Programs. In *European Symp. on Programming.*

[6] P. Billingsley. 2012. *Probability and Measure.* John Wiley & Sons, Inc.

[7] J. Borgström, U. D. Lago, A. D. Gordon, and M. Szymczak. 2016. A Lambda-Calculus Foundation for Universal Probabilistic Programming. In *Int. Conf. on Functional Programming.*

[8] K. Chatterjee, H. Fu, P. Novotný, and R. Hasheminezhad. 2016. Algorithmic Analysis of Qualitative and Quantitative Termination Problems for Affine Probabilistic Programs. In *Princ. of Prog. Lang.*

[9] K. Chatterjee, P. Novotný, and Đ. Žikelić. 2017. Stochastic Invariants for Probabilistic Termination. In *Princ. of Prog. Lang.*

[10] P. Cousot and R. Cousot. 1977. Abstract Interpretation: A Unified Latice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Princ. of Prog. Lang.*

[11] P. Cousot and R. Cousot. 1979. Systematic Design of Program Analysis Frameworks. In *Princ. of Prog. Lang.*

[12] J. I. den Hartog and E. P. de Vink. 1999. Mixing Up Nondeterminism and Probability: a preliminary report. *Electr. Notes in Theor. Comp. Sci.* (1999).

[13] E. W. Dijkstra. 1997. *A Discipline of Programming.* Prentice Hall PTR Upper Saddle River, NJ, USA.

[14] T. Ehrhard, M. Pagani, and C. Tasson. 2018. Measurable Cones and Stable, Measurable Functions. In *Princ. of Prog. Lang.*

[15] A. Farzan and Z. Kincaid. 2015. Compositional Recurrence Analysis. In *Formal Methods in Computer-Aided Design.*

[16] L. M. Ferrer Fioriti and H. Hermanns. 2015. Probabilistic Termination: Soundness, Completeness, and Compositionality. In *Princ. of Prog. Lang.*

[17] G. Gallo, G. Longo, S. Pallottino, and S. Nguyen. 1993. Directed Hypergraphs and Applications. *Disc. Appl. Math.* (1993).

[18] Z. Ghahramani. 2015. Probabilistic machine learning and artificial intelligence. *Nature* (2015).

[19] A. D. Gordon, T. A. Henzinger, A. V. Nori, and S. K. Rajamani. 2014. Probabilistic Programming. In *Future of Softw. Eng.*

[20] C. A. Gunter, P. D. Mosses, and D. S. Scott. 1989. *Semantic Domains and Denotational Semantics.* Technical Report. University of Pennsylvania Department of Computer and Information Science.

[21] N. Jansen, B. L. Kaminski, J.-P. Katoen, F. Olmedo, F. Gretz, and A. K. McIver. 2015. Conditioning in Probabilistic Programming. *Electr. Notes in Theor. Comp. Sci.* (2015).

[22] C. Jones. 1989. *Probabilistic Non-determinism.* Ph.D. Dissertation. University of Edinburgh Edinburgh.

[23] C. Jones and G. Plotkin. 1989. A Probabilistic Powerdomain of Evaluations. In *Logic in Computer Science.*

[24] B. L. Kaminski, J.-P. Katoen, C. Matheja, and F. Olmedo. 2016. Weakest Precondition Reasoning for Expected Run—Times of Probabilistic Programs. In *European Symp. on Programming.*

[25] M. Kattenbelt, M. Kwiatkowska, G. Norman, and D. Parker. 2009. Abstraction Refinement for Probabilistic Software. In *Verif., Model Checking, and Abs. Interp.*

[26] J. Knoop and B. Steffen. 1992. The Interprocedural Coincidence Theorem. In *Comp. Construct.*

[27] D. Kozen. 1981. Semantics of Probabilistic Programs. *J. Comput. Syst. Sci.* (1981).

[28] D. Kozen. 1985. A Probabilistic PDL. *J. Comput. Syst. Sci.* (1985).

[29] A. Lal, T. Touili, N. Kidd, and T. Reps. 2008. Interprocedural Analysis of Concurrent Programs Under a Context Bound. In *Tools and Algs. for the Construct. and Anal. of Syst.*

[30] A. K. McIver and C. C. Morgan. 2001. Partial correctness for probabilistic demonic programs. *Theor. Comp. Sci.* (2001).

[31] A. K. McIver and C. C. Morgan. 2005. *Abstraction, Refinement and Proof for Probabilistic Systems.* Springer Science+Business Media, Inc.

[32] M. Mislove. 2000. Nondeterminism and Probabilistic Choice: Obeying the Laws. In *Concurrency Theory.*

[33] M. Mislove, J. Ouaknine, and J. Worrell. 2004. Axioms for Probability and Nondeterminism. *Electr. Notes in Theor. Comp. Sci.* (2004).

[34] M. Müller-Olm and H. Seidl. 2004. Precise Interprocedural Analysis through Linear Algebra. In *Princ. of Prog. Lang.*

[35] F. Olmedo, B. L. Kaminski, J.-P. Katoen, and C. Matheja. 2016. Reasoning about Recursive Probabilistic Programs. In *Logic in Computer Science.*

[36] P. Panangaden. 1999. The Category of Markov Kernels. *Electr. Notes in Theor. Comp. Sci.* (1999).

[37] S. Smolka, P. Kumar, N. Foster, D. Kozen, and A. SIlva. 2017. Cantor meets Scott: Semantic Foundations for Probabilistic Networks. In *Princ. of Prog. Lang.*

[38] S. Staton. 2017. Commutative Semantics for Probabilistic Programming. In *European Symp. on Programming.*

[39] S. Staton, H. Yang, C. Heunen, and O. Kammar. 2016. Semantics for probabilistic programming: higher-order functions, continuous distributions, and soft constraints. In *Logic in Computer Science.*

[40] R. Tix, K. Keimel, and G. Plotkin. 2009. Semantic Domains for Combining Probability and Non-Determinism. *Electr. Notes in Theor. Comp. Sci.* (2009).

[41] D. Wang, J. Hoffmann, and T. Reps. 2017. PMAF: An Algebraic Framework for Static Analysis of Probabilistic Programs. Available on: http://www.cs.cmu.edu/~janh/papers/WangHR17.pdf. (2017).

[5] A dual notion of probabilistic *over*-abstraction is also defined and the corresponding soundness theorem is proved. We omit these due to the space limit.