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
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 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., Bernoulli(0.5), where Bernoulli(\( \rho \)) represents a coin flip that returns true with probability \( \rho \). 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

\[
\text{if } \ ● \ \text{then } A \ \text{else } B \ \text{fi}
\]

where ● represents a nondeterministic choice. We can view Bernoulli(0) and Bernoulli(1) as two candidates for the refinement of ●, 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 ● 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 \( X \rightarrow X \) then the nondeterminism-last approach defines semantics with the signature \( X \rightarrow \mathcal{P}(X) \), while nondeterminism-first leads to \( \mathcal{P}(X \rightarrow X) \), where \( \mathcal{P}(S) \) 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...
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) ∧ n ≥ 10. The collection of all such mixed choices is called *deterministic choices*. The *sampling statement* x ~ 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.

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

if Bernoulli(0.9) do
  n := n + 1;
else if n ≥ 10 then break
  ⋆ then x := x + z
else y := y + z
fi

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

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 → P(E, ×; 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 Börgström 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 weakestprecondition 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 \( X \rightarrow \mathcal{P}(\mathcal{D}(X)) \), 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 \S 1, we are interested in developing a semantics that resolves nondeterminism first, and then input. Intuitively, the form of our semantics should be \( \mathcal{P}(X \rightarrow \mathcal{D}(X)) \).

### 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 \((V, E, e^{\text{entry}}, e^{\text{exit}})\), where \( V \) is a finite set of nodes, \( E \) is a set of hyper-edges, \( e^{\text{entry}} \in V \) is a distinguished entry node, and \( e^{\text{exit}} \in V \) is a distinguished exit node. A hyper-edge is an ordered pair \((x, Y)\), where \( x \in V \) is a node and \( Y \subseteq V \) is an ordered, non-empty set of nodes. For a hyper-edge \( e = (x, Y) \) in \( E \), we use \( \text{src}(e) \) to denote \( x \) and \( \text{Dst}(e) \) 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 \( e^{\text{entry}} \) has no incoming edges, and \( e^{\text{exit}} \) has no outgoing edges.

**Definition 3.2** (Probabilistic programs). A probabilistic program contains a finite set of procedures \( \{(H_i, LV_i)\}_{1 \leq i \leq n} \), where each procedure is a pair where \( H_i = (E_i, v_i^{\text{entry}}, v_i^{\text{exit}}) \) is a control-flow hyper-graph in which each node except \( v_i^{\text{exit}} \) has at least one outgoing hyper-edge, \( v_i^{\text{exit}} \) has no outgoing edge, and \( LV_i \) is a finite set of local variables. We assume 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 \( e \in E = \bigcup_{1 \leq i \leq n} E_i \) a

\[
\mathcal{A} ::= x := e \mid x \sim \mathcal{D} \mid \text{observe}(\phi) \mid \text{skip}
\]

\( \phi \in \mathcal{L} ::= \text{true} \mid \text{false} \mid e \leftrightarrow u \mid \mathcal{F}(\varnothing, \leq, \leq) \mid \neg \phi \mid \mathcal{F} \)

where \( e, u \in \text{Exp} = x \mid e \in \mathbb{R} \mid \bullet \bullet u \mid \bullet \bullet \bullet \bullet \bullet \)

\( x \in \text{Var} \subseteq \mathbb{N} \cup \mathbb{N} \cup \mathbb{R} \ni \mathcal{V}_i := x \mid y \mid z \mid \cdots \)

\( \mathcal{F} \mathcal{D} \in \mathcal{D} \text{Dist} \triangleq \text{Bernoulli}(\epsilon) \mid \text{sampleBool}(\mathcal{F}) \)

\( \mathcal{D} \in \mathcal{D} \triangleq \text{Uniform}(e, u) \mid \text{Gaussian}(e, u) \mid \text{sampleReal}(\mathcal{F}) \)

**Figure 3.** Examples of data actions and deterministic conditions

**control-flow action** \( \text{Ctrl}(e) \), where \( \text{Ctrl} \) is

\[
\text{Ctrl} ::= \text{seq}([\text{act}]) \mid \text{cond}(\phi) \mid \text{call}([i \mapsto j]) \mid \text{observe}(\phi) \mid \text{skip}
\]

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

See Fig. 3 for data actions \( \mathcal{A} \) and deterministic conditions \( \mathcal{L} \) that would be used for an arithmetic program. \( \text{sampleBool}(\mathcal{F}) \) samples a Boolean value with respect to the probability density function \( f \), where \( \mathcal{F} \) is a vector of parameters, while \( \text{sampleReal}(\mathcal{F}) \) samples a real value. For example, \( \text{Gaussian}(e, u) \) can be represented as \( \text{sampleReal}(e, u) \) where \( f = \lambda(\mu, \sigma), \lambda(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \). Note that \( \text{goto}, \text{break}, \text{and continue} \) 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 \( (v_2, \{v_3, v_4\}) \) is associated with a sequencing action \( \text{seq}([n := n + 1]) \), while \( (v_1, \{v_2, v_4\}) \) is assigned a deterministic-choice action \( \text{cond}(\text{Bernoulli}(0.9)) \).

#### 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 \((X, \Sigma)\) where \( X \) is a non-empty set called the sample space, and \( \Sigma \) is a \( \sigma \)-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 \((X_1, \Sigma_1)\) to a measurable space \((X_2, \Sigma_2)\) is a mapping \( f : X_1 \rightarrow X_2 \) such that for all \( A \in \Sigma_2, f^{-1}(A) \in \Sigma_1 \). The measurable functions from a measurable space \((X, \Sigma)\) to a Borel space \( \mathcal{B}(\mathbb{R}_{>0}) \) on nonnegative real numbers (the smallest \( \sigma \)-algebra containing all open intervals) are called \( \mathcal{L} \)-measurable.

A measure \( \mu \) on a measurable space \((X, \Sigma)\) is a function from \( \Sigma \) 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 \( i \neq j \) we have \( \sum_{i=1}^{\infty} \mu(A_i) = \mu(\bigcup_{i=1}^{\infty} A_i) \)). The measure \( \mu \) is called a (sub-probability) distribution if \( \mu(X) \leq 1 \). If \( \mu \) is a distribution and \( e \in [0, 1] \), we write \( c \cdot \mu \) for the distribution \( \lambda A.e \cdot \mu(A) \). If \( \mu, \nu \) are distributions and \( c, d \in [0, 1] \) such that \( c + d \leq 1 \), 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
Suppose that $P = \langle V, E, \tau^{\text{entry}}, \tau^{\text{exit}} \rangle$ is a single-procedure deterministic program. Therefore, each node in $P$ except $\tau^{\text{exit}}$ is associated with exactly one hyper-edge. The program configurations $T = V \times \Omega$ are pairs of the form $(v, \omega)$, 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 $\forall \tau^a \in (V, \tau^b)$ is naturally a measurable space, we define $\mathcal{F}$ as the product space of $\mathcal{Y}$ and $\mathcal{D}$ to be a measurable space over program configurations $T$.

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

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

\begin{align*}
\langle v_0, n \rightarrow 233 \rangle & \rightarrow_{\lambda A.([v_2, n \rightarrow 0] \in A)} \langle v_1, n \rightarrow 1 \rangle \rightarrow_{\lambda A.0.9 \cdot ([v_2, n \rightarrow 1] \in A) + 0.1 \cdot ([v_4, n \rightarrow 1] \in A)} \langle v_2, n \rightarrow 9 \rangle \\
\langle v_1, n \rightarrow 9 \rangle & \rightarrow_{\lambda A.([v_1, n \rightarrow 9] \in A)}
\end{align*}

Lemma 3.5. $\rightarrow_n$ is a kernel.

Then we define step-indexed evaluation as the family of $n$-indexed relations $\langle v, \omega \rangle \rightarrow_{n} \mu \rightarrow (v, \omega)$ and distributions $\mu$ on program states inductively, as shown in Fig. 7.

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

\begin{align*}
\langle v_4, n \rightarrow 10 \rangle & \rightarrow_{\lambda F.([n \rightarrow 10] \in F)} \langle v_2, n \rightarrow 0 \rangle \\
\langle v_2, n \rightarrow 0 \rangle & \rightarrow_{\lambda F.0.1 \cdot ([n \rightarrow 0] \in F)} \langle v_1, n \rightarrow 0 \rangle \\
\langle v_1, n \rightarrow 0 \rangle & \rightarrow_{\lambda F.0.1 \cdot ([n \rightarrow 0] \in F) + 0.09 \cdot ([n \rightarrow 1] \in F)} \langle v_0, n \rightarrow 10 \rangle
\end{align*}

Lemma 3.7. $\rightarrow_n$ is a kernel for all $n$.

Because the set of distributions with the pointwise order forms an $\omega$-cpo (i.e., an $\omega$-complete partial order), we define the semantics of the program $P$ as $\langle P \rangle_{\omega} \triangleq \lambda \omega. \sup_{n \in \mathbb{N}} \mu(\langle \tau^{\text{entry}}, \omega \rangle \rightarrow_{n} \mu)$.

Example 3.8. In Fig. 4, $\langle P \rangle_{\omega}(s)$ for any initial state $s$ is given by

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

Theorem 3.9. $\langle P \rangle_{\omega}$ 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., \( z > 1 \land Bernoulli(0.5) \)), 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 \( \xrightarrow{e} \). Let \( \mathcal{P} = (\Omega, \mathcal{F}) \) be the measurable space over program states. We define a scheduler \( \sigma \) as a measurable function \( \Omega \rightarrow [0, 1] \). A scheduler trace \( t \) is a finite sequence \( \{\sigma_1, \ldots, \sigma_m\} \) of schedulers.

We redefine step-indexed evaluations as \( \xrightarrow{t} \), where \( t \) is a scheduler trace, in Fig. 8.

\[
\begin{align*}
\langle v, \omega \rangle & \xrightarrow{0} 0 \\
\langle v^{\text{exit}}, \omega \rangle & \xrightarrow{t} n \delta(\omega) \quad \text{if } n > 0 \\
\langle v, \omega \rangle & \xrightarrow{t+1} \mathcal{L}F_t \int \mu_t(F)(\mu(dr)) \\
& \text{where } v \text{ has 1 outgoing edge } e, \langle v, \omega \rangle \xrightarrow{e} \mu \\
& \text{and } t \xrightarrow{1} n \mu_t \text{ for any } t \in \supp(\mu) \\
\langle v, \omega \rangle & \xrightarrow{t+1} \sigma(\omega) \cdot \mu_1 + (1 - \sigma(\omega)) \cdot \mu_2 \\
& \text{where } v \text{ has 2 outgoing edges } e_1, e_2 \\
& \text{and } \langle v, \omega \rangle \xrightarrow{\mathcal{L}F_t \int e_1(F)(\mu(dr))} \mu_t \text{ for any } \tau \in \supp(\nu_j) \\
\end{align*}
\]

**Figure 8.** Annotated step-indexed evaluation relation

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

\[
S_m = \{ \lambda \omega. \sup \{ \mu \mid \langle v^{\text{entry}}, \omega \rangle \xrightarrow{t} n \mu \mid t \in \mathcal{T} \} \}
\]

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 \( \vee D \). \( X \) is called directed complete or a dcpo if each directed subset \( D \) has a supremum \( \vee D \) in \( X \). A dcpo \( X \) is pointed if \( X \) contains a least element. A function \( f : X \rightarrow Y \) between two dcpos is Scott-continuous if it is monotone and preserves directed joins, i.e., \( f(\vee D) = \vee f(D) \) for all directed subsets \( D \). The set of all Scott-continuous functions between \( X \) and \( Y \) is denoted as \( [X \rightarrow Y] \).

The lower closure of a subset \( A \) is defined as \( \downarrow A \overset{\text{def}}{=} \{ x \in X \mid \exists a \in A.x \leq a \} \). The upper closure of a subset \( A \) is defined as \( \uparrow A \overset{\text{def}}{=} \{ x \in X \mid \exists a \in A.a \leq x \} \). 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 \( \vee D \in A \) for every directed subset \( D \subseteq A \). The complement \( \mathcal{X} \setminus A \) 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 \( O(X) \). A function \( f : X \rightarrow Y \) between two topological spaces is topologically continuous if for all \( U \in O(Y), f^{-1}(U) \in O(X) \). 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 \( X \), \( y \leq \vee D \) implies \( x \leq d \) for some \( d \in D \). We define \( \overset{\text{def}}{=} \{ x \in X \mid \exists a \in A.x \ll a \} \) and \( \overset{\text{def}}{=} \{ x \in X \mid \exists a \in A.a \ll x \} \). \( X \) is called continuous if for all \( x \) of \( X \), the set \( \ll x \) is directed and \( x = \vee \ll x \). 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 members 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 an 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., in [23, 40]), and extend the ideas to kernels.

Let \( X \) be a dcpo. A function \( \mu : O(X) \rightarrow [0, 1] \) is called a valuation on \( X \) if: (i) \( \mu(\emptyset) = 0 \), (ii) \( U \subseteq V \) implies \( \mu(U) \leq \mu(V) \) for all \( U, V \in O(X) \), and (iii) \( \mu(U) + \mu(V) = \mu(U \cup V) + \mu(U \cap V) \) for all \( U, V \in O(X) \). If \( \mu \) 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 \( O(X) \), then \( \mu \) is called a continuous valuation. The set of all continuous valuations on \( X \) is denoted by \( D(X) \). 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 \( \mu \) 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 $\kappa: [X \to D(X)]$ is called a transition map. Let $\mathcal{K}(X) \overset{\text{def}}{=} [X \to D(X)]$. Then $\mathcal{K}(X)$ is naturally a dcpo with the pointwise order, denoted by $\subseteq$. We will show the definition of transition maps is reasonable with respect to that of kernels.

**Lemma 4.1.** Let $\kappa$ be a continuous function in $[X \to D(X)]$.
- 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 $\kappa$ as $\int f(y)\kappa(x)(dy)$. We then define composition of transition maps $\kappa_1, \kappa_2$ as $\kappa_1 \circ \kappa_2 \overset{\text{def}}{=} \lambda x . \lambda U. \int \kappa_1(x)(dy)\kappa_2(y)(U)$, and conditional choice of transition maps $\kappa_1, \kappa_2$ conditioning on $f$ as $\kappa_1 f \circ \kappa_2 \overset{\text{def}}{=} \lambda x . f(x) \cdot \kappa_1(x) + (1-f(x)) \cdot \kappa_2(x)$, 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.
- $\Omega$ is a dcpo over program states, with a Scott topology. For arithmetic programs, the state space $\Omega$ is defined as $\forall a \to (\mathbb{R} \cup \{+\infty\})$, with the pointwise order.
- Data actions are interpreted as transition maps $\overrightarrow{act} \in \mathcal{K}(\Omega)$. For simplicity, suppose the program only has one variable and $\Omega$ 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 $\delta_a$ is represented as $\lambda a. \lambda x.a \cdot \mathbb{I}(x>a)$. The distribution with probability density function $f$ can be denoted as $\lambda a. \int_{x>a} f(x)dx$, 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 $\Omega$ 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 $\llbracket P \rrbracket_\text{ops}$ for it can be derived as a transition map. We will override $\llbracket P \rrbracket_\text{ops}$ to denote the domain-theoretic operational semantics in the following sections.

Here we state a useful lemma:

**Lemma 4.3.** $\otimes$ and $f \circ$ 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 $\varphi$ and $B$ establishes $\psi$, then a nondeterministic choice of $A$ and $B$ should include a semantic object that establishes $\varphi$ with probability 0.3 and $\psi$ 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 $\{ p \cdot a + (1-p) \cdot b \mid a \in A \land b \in B \land 0 \leq p \leq 1 \}$. This set is the convex combination of $A$ and $B$. On the other hand, the nondeterministic choice of $A$ and $B$ 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 $f \circ$ operator for every feasible function $f$ from $X$ to $[0, 1]$. Recall that the definition $\kappa_1 f \circ \kappa_2 = \lambda x. f(x) \cdot \kappa_1(x) + (1-f(x)) \cdot \kappa_2(x)$ 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 $\kappa_1 f \circ \kappa_2$ 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 $(a \circ b \mid a \in S_1 \land b \in S_2)$ 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 $\text{conv}(A)$.

**Example 4.6.** Suppose $X = \mathbb{R} \cup \{+\infty\}$. Let $\kappa_1 \overset{\text{def}}{=} \lambda x.\lambda (a, +\infty).[x + 1 > a]$ and $\kappa_2 \overset{\text{def}}{=} \lambda x.\lambda (a, +\infty).[x - 1 > a]$. Intuitively, $\kappa_1$ and $\kappa_2$ 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. $\text{conv}(\{\kappa_1, \kappa_2\})$ 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 $x := x - 1$ should at least model the meaning of $\text{conv}(\{\kappa_1, \kappa_2\})$.

Following are some properties of the $\text{conv}(\cdot)$ operator.

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

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

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

**Lemma 4.9.** For an arbitrary $S \subseteq \mathcal{K}(X)$, we have $\text{conv}(S) = \bigcup_{F \subseteq S, F \text{ finite}} \text{conv}(F)$.

### 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
\( \Psi \) 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 \( x \supseteq x \cup y \).

There is an interesting connection between powerdomain constructions and partial/total correctness used in program verification [20]. Intuitively, the formal union \( \Psi \) 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 \( \varphi \) under partial correctness, then \( A \cup B \) should also satisfy \( \varphi \) for any \( B \), i.e., the nondeterministic-choice behaves angelically. On the other hand, if a program \( A \cup B \) satisfies \( \varphi \) under total correctness, then both \( A \) and \( B \) must satisfy \( \varphi \), 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.
- \( x \supseteq x \cup y \) coincides with total correctness, hence the Smyth construction stands for total correctness.

**Example 5.1.** Consider the following three programs:

\[
\begin{align*}
\text{\( x \sim \text{Uniform}(0,1) \) if \( \text{\( \star \)} \) then \( x \sim \text{Uniform}(0,1) \) else \( \text{observe(false)} \) fi} \\
\text{observe(false)}
\end{align*}
\]

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—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 \( A \leq_{EM} B \), if the following conditions hold: (i) for all \( a \in A \), there exists \( b \in B \) such that \( a R b \), and (ii) for all \( b \in B \), there exists \( a \in A \) such that \( a R b \).

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 \( \mathcal{P}X \). Then we want to lift composition \( \otimes \) and conditional-choice \( f \) to the powerdomain, denoted as \( \otimes_{\mathcal{P}} \) and \( f_{\mathcal{P}} \), 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 \( A \) and \( B \).
- 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 \( \Psi_{\mathcal{P}} \) on \( \mathcal{P}X \).

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

**Conditional-choice.** For two elements \( A, B \) of \( \mathcal{P}X \), and a continuous function \( f \) from \( X \) to \([0,1]\), \( A \otimes_{\mathcal{P}} B \) should contain all pairwise conditional-choices of transition maps in \( A \) and \( B \), i.e., \( \{ \kappa_1 \otimes \kappa_2 \mid \kappa_1 \in A \land \kappa_2 \in B \} \). 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.

\[
\begin{align*}
A \otimes_{\mathcal{P}} B &= A \\
A \otimes_{\mathcal{P}} B &= B \otimes_{\mathcal{P}} A \\
(A_f \otimes_{\mathcal{P}} B)_{g \otimes_{\mathcal{P}} C} &= A_{f \circ g} \otimes_{\mathcal{P}} (B_{g \circ h} \otimes_{\mathcal{P}} C) \\
\text{where } f' &= f \cdot g \text{ and } (1 - f') \cdot (1 - g') = 1 - g
\end{align*}
\]

**Figure 9.** Laws for conditional-choice

### 5.3 The Hoare Construction

Let \( X \) be a dcpo. We consider the collection \( \mathcal{H}X \) of \( \{ S \subseteq \mathcal{K}(X) \mid S \text{ nonempty, generally convex, Scott-closed} \} \) ordered by \( \sqsubseteq \). We denote \( B \equiv A \sqsubseteq B \), which is equivalent to inclusion \( \sqsubseteq \).

**Lemma 5.3.** (\( \mathcal{H}X, \sqsubseteq \)) is a dcpo.
The bottom element in $\mathbb{H}X$ is defined as $\bot_{\mathbb{H}} \overset{\text{def}}{=} \{\lambda x. U \cdot 0\}$. The directed suprema in $\mathbb{H}X$ is performed as $\vee A_i = \bigcup A_i$. Then we can lift kernel composition $\otimes$ and conditional-choice $\phi$ to the powerdomain $\mathbb{H}X$.

$$A \otimes B \overset{\text{def}}{=} \text{conv}(a \otimes b \mid a \in A \land b \in B)$$

$$A \phi \otimes B \overset{\text{def}}{=} \{a \phi b \mid a \in A \land b \in B\}$$

**Lemma 5.4.** $\otimes$ and $\phi$ are Scott-continuous, $(\mathbb{H}X, \otimes, \bot_{\mathbb{H}})$ is a monoid where $\bot_{\mathbb{H}} \overset{\text{def}}{=} \bot_{\mathbb{H}} \subseteq \{\lambda x. \lambda \cdot \} \cdot \{x \in U\}$, and laws in Fig. 9 hold for $\phi\otimes$.

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

**Lemma 5.5.** $\mathbb{H}$ 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 $x \sim \text{Uniform}(0, 1)$ and $\text{observe}(\text{false})$ 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 $\{\kappa_1\}$ and $\{\kappa_2\}$ in $\mathbb{H}X$, respectively. Note that $\{\kappa_2\}$ is actually $\bot_{\mathbb{H}}$. Because $\kappa_2 \nsubseteq \kappa_1$, we know that $\{\kappa_2\} \nsubseteq \{\kappa_1\}$. The semantics of program (3) is then derived as $\{\kappa_1\} \cup \{\kappa_2\} = \text{conv}(\{\kappa_1\} \cup \{\kappa_2\}) \subseteq \{\kappa_1\}$, which is the same as the semantics of program (2).

**5.4 The Smyth Construction**

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

Let $X$ be a dcpo and $f : X \rightarrow X$ be a continuous function. $f$ is finitely separate from the identity on $X$ if there exists a finite set $M$ such that for any $x \in X$ there is $m \in M$ with $f(x) \leq m \leq x$. 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 $[X \rightarrow Y]$ is continuous [1, Prop. 4.2.10].

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

We consider the collection

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

ordered by $A \subseteq B \overset{\text{def}}{=} A \supseteq B$, which is equivalent to reverse inclusion $\supseteq$.

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

The bottom element in $\mathbb{S}X$ is defined $\bot_{\mathbb{S}} = \mathcal{K}(X)$. The directed suprema in $\mathbb{S}X$ is performed as $\vee A_i = \bigcap A_i$. Then we can lift kernel composition $\otimes$ and conditional-choice $\phi$ to the powerdomain $\mathbb{S}X$.

$$A \otimes B \overset{\text{def}}{=} \uparrow \text{conv}(a \otimes b \mid a \in A \land b \in B)$$

$$A \phi \otimes B \overset{\text{def}}{=} \{a \phi b \mid a \in A \land b \in B\}$$

**Lemma 5.9.** $\otimes$ and $\phi$ are Scott-continuous, $(\mathbb{S}X, \otimes, \bot_{\mathbb{S}})$ is a monoid where $\bot_{\mathbb{S}} \overset{\text{def}}{=} \{\lambda x. \lambda U \cdot [x \in U]\}$, and the laws in Fig. 9 hold for $\phi\otimes$.

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

**Lemma 5.10.** $\cup$ 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 $\{\kappa_2\}$ in $\mathbb{S}X$, respectively. Note that $\{\kappa_2\}$ is actually $\bot_{\mathbb{S}}$. Because $\kappa_2 \nsubseteq \kappa_1$, we know that $\{\kappa_2\} \nsubseteq \{\kappa_1\}$. The semantics of program (3) is then derived as $\{\kappa_1\} \cup \{\kappa_2\} = \uparrow \text{conv}(\{\kappa_1\} \cup \{\kappa_2\}) \subseteq \{\kappa_1\}$, 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 formulated 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 $v$—or 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 $\mathcal{D} = \Omega$ with a Scott topology over program states.
- A mapping from data actions act to transition maps act $\in \mathcal{K}(\Omega)$.
- A mapping from logical conditions $\phi$ to continuous functions $\phi \in \{\Omega \rightarrow [0, 1]\}$.

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

- if $e = (v, \{u_1, \ldots, u_k\}) \in E$, then $\mathcal{S}(v) = \text{Ctrl}(\mathcal{S}(u_1), \ldots, \mathcal{S}(u_k))$, and
- otherwise, $\mathcal{S}(v) = \lambda \omega. \lambda F. \lambda\omega \in F$.

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

$$\text{seq}(\text{act})(\kappa_1) \overset{\text{def}}{=} \text{act} \otimes \kappa_1$$

$$\text{cond}(\phi)(\kappa_1, \kappa_2) \overset{\text{def}}{=} \kappa_1 \phi \otimes \kappa_2$$
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:

$$
\lambda S.\lambda v. \begin{cases} 
\text{Ctrl}(e)(S(u_1), \ldots, S(u_k)) & e = (v, (u_1, \ldots, u_k)) \in E \\
\lambda \alpha. \lambda F. [v \in F] & \text{otherwise}
\end{cases}
$$

Recall Keene’s fixed point theorem:

**Theorem 6.2.** Suppose $\langle X, \leq \rangle$ is a dcpo with a least element $\bot$, and let $f : X \rightarrow X$ be a Scott-continuous function. Then $f$ has a least fixed point, denoted by $\lambda f^\omega$. By the Scott-continuity of $\otimes$ and $\diamondsuit$ (stated in Lemma 4.3), we derive the Scott-continuity of $F_P$.

**Lemma 6.3.** The function $F_P$ is Scott-continuous on $\langle V \rightarrow \mathcal{K}(\Omega, \subseteq) \rangle$, which is a dcpo with the least element $\lambda v. \bot$. Hence we define the semantics of the program $P$ as $[P]_{ds} \overset{df}{=} (\lambda f^\omega, F_P)(\nu^{entry})$. We can then prove its equivalence to the operational semantics $[P]_{ao}$ described in Ex. 4.2.

**Theorem 6.4.** $[P]_{ao} = [P]_{ds}$.

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 have at least one outgoing edge. Recursion is introduced by enabling multiple procedures and the calling actions $call[i \rightarrow j]$, 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 $\mathbb{P}H\Omega$ or $\mathbb{P}S\Omega$, depending on if we want to reason about partial correctness or total correctness, respectively. Given a probabilistic program $P = \langle H_i \rangle_{1 \leq i \leq n}$, let $S(v) \in \mathbb{P}(\Omega)$ be the semantics assigned to the node $v$; the following local properties should hold:

- if $v \notin v_i^{exit}$, then $S(v) = \bigcup_{e = (v, (u_1, \ldots, u_k)) \in E} \text{Ctrl}(e)(S(u_1), \ldots, S(u_k))$, and
- otherwise, $S(v) = \bot$.

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

$$
\begin{align*}
\text{seq}(\text{act})(S_1) \overset{df}{=} P(\text{act}) \otimes_{\mathbb{P}} S_1 \\
\text{cond}[\varphi](S_1, S_2) \overset{df}{=} S_1 \varphi \otimes_{\mathbb{P}} S_2 \\
\text{call}[i \rightarrow j](S_1) \overset{df}{=} S(v_i^{entry}) \otimes_{\mathbb{P}} S_1
\end{align*}
$$

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

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

$$
\lambda S.\lambda v. \begin{cases} 
\text{Ctrl}(e)(S(u_1), \ldots, S(u_k)) & e = (v, (u_1, \ldots, u_k)) \in E \\
\lambda \alpha. \lambda F. [v \in F] & \text{otherwise}
\end{cases}
$$

**Lemma 6.5.** The function $F_P$ is Scott-continuous on $(V \rightarrow \mathbb{P}(\mathcal{K}(\Omega, \subseteq))$, which is a dcpo with the least element $\lambda v. \bot$.

6.3 Local Variables

We can extend the semantics in §6.2 to handle local variables following a standard approach introduced by Knopf 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 $i \rightarrow j$, the values of local variables in $LV_i$ are recorded and inaccessible to $H_j$ and procedures transitively called by $H_i$—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 $\text{Project}(x)(A)$, where the following hold, for any $\ast \in \{\otimes, \diamondsuit, \varnothing, \cup\}$:

$$
\begin{align*}
\text{Project}(x)(\text{Project}(y)(A)) & = \text{Project}(y)(\text{Project}(x)(A)) \\
\text{Project}(x)(\text{Project}(x)(A) \ast B) & = \text{Project}(x)(A) \ast \text{Project}(x)(B) \\
\text{Project}(x)(A) \ast \text{Project}(y)(A) & = \text{Project}(x)(y) \ast \text{Project}(x)(B)
\end{align*}
$$

We also require the operator $\text{Project}(x)$ to be Scott-continuous.

With the projection operator, we can interpret the calling action as $call[i \rightarrow j](S_1) \overset{df}{=} \text{Project}(LV_i)(S(v_i^{entry})) \otimes_{\mathbb{P}} \bot$. Because $\text{Project}(x)$ 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 $\mathbb{V}$ be the set of program variables; the set of program states is $\Omega = \mathbb{V} \rightarrow (\mathbb{R} \cup \{+\infty\})$. Then for a variable $x \in \mathbb{V}$ in Var and transition map $\kappa$, we define $3x(\kappa) \overset{df}{=} \lambda \alpha. \lambda F. (\kappa \otimes x = \omega(x)(\kappa))(F)$. For Hoare construction, we define $3\text{Project}_{\mathbb{V}}(x)(A) \overset{df}{=} \{3x(\kappa) \mid x \in A\}$. For Smyth construction, we define $\text{Project}_{\mathbb{V}}(x)(A) \overset{df}{=} \{3x(\kappa) \mid \kappa \in A\}$.

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

- $3x(\cdot)$ is a Scott-continuous operator on $\mathbb{K}(\Omega)$.
- $\text{Project}_{\mathbb{V}}(x)(\cdot)$ is a Scott-continuous operator on $\mathbb{P}\mathbb{V}$.
- $\text{Project}_{\mathbb{V}}(x)(\cdot)$ is a Scott-continuous operator on $\mathbb{S}\mathbb{V}$.

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 $L$ is a 7-tuple $M = (M, \subseteq, \otimes, \varnothing, \cup, \bot, 1)$, where $(M, \subseteq, \otimes, 1)$ forms an $\omega$-cpo with a least element $\bot$, $(M, \otimes, \cup)$ forms a monoid (i.e., $\varnothing$ is an associative binary operator with $1$ as its identity element); $\otimes$ is a binary operator parametrized by $\varnothing$ which is a condition in $L$; $\cup$ is a binary operator...
that is idempotent, commutative, and associative; $\otimes, \varphi, \varphi$ and $\cup$ are pre-$\omega$-continuous and the following properties hold:

\[
\begin{align*}
\varphi \oplus b & \subseteq a \cup b & a \subseteq a \otimes b & a \cup a = a & a \otimes b & = b \otimes a \end{align*}
\]

\[(\varphi \oplus b) \cup c = a \otimes (b \cup c) \quad \text{where} \quad \varphi' = \varphi \wedge \varphi, \varphi' \vee \varphi' = \varphi'
\]

The precedence of the operators is that $\otimes$ binds tightest, followed by $\oplus$ and $\cup$.

**Lemma 7.2.** The denotational semantics in §6.2 is a PMA $C = (\mathcal{P}_C, \mathcal{P}_A, \cdot, \circ, \exists, \mathcal{I}_V, \mathcal{I}_A, \mathcal{L}_C)$. Theorem 7.5. Let $\lambda \vdash \psi \circ a \cup a$; then for all data actions $a$, $\langle \mathcal{I}_A \rangle \subset \mathcal{I}_C \cup \mathcal{I}_V$.

**References**


