PMAF: An Algebraic Framework for Static Analysis of Probabilistic Programs

Di Wang[1], Jan Hoffmann[1], and Thomas Reps[2,3]

[1]Carnegie Mellon University
[2]University of Wisconsin
[3]GrammaTech, Inc.

Abstract

Automatically establishing that a probabilistic program satisfies some property φ is a challenging problem. While a sampling-based approach—which involves running the program repeatedly—can *suggest* that φ holds, to establish that the program *satisfies* φ, analysis techniques must be used. Despite recent successes, probabilistic static analyses are still more difficult to design and implement than their deterministic counterparts. This paper presents a framework, called *PMAF*, for designing, implementing, and proving the correctness of static analyses of probabilistic programs with challenging features such as recursion, unstructured control-flow, divergence, nondeterminism, and continuous distributions. PMAF introduces *pre-Markov algebras* to factor out common parts of different analyses. To perform *interprocedural analysis* and to create *procedure summaries*, PMAF extends ideas from non-probabilistic interprocedural dataflow analysis to the probabilistic setting. One novelty is that PMAF is based on a semantics formulated in terms of a control-flow *hyper-graph* for each procedure, rather than a standard control-flow graph. To evaluate its effectiveness, PMAF has been used to reformulate and implement existing *intra*procedural analyses for Bayesian-inference and the Markov decision problem, by creating corresponding *inter*procedural analyses. Additionally, PMAF has been used to implement a new interprocedural *linear expectation-invariant analysis*. Experiments with benchmark programs for the three analyses demonstrate that the approach is practical.

Keywords— Program analysis, probabilistic programming, expectation invariant, pre-Markov algebra

1 Introduction

Probabilistic programming is becoming increasingly popular because it provides a rich framework for implementing randomized algorithms [Barthe et al. 2016b], cryptography protocols [Barthe et al. 2009], cognitive [Gordon et al. 2014] models, and machine learning [Ghahramani 2015] algorithms. Static analysis of probabilistic programs has received a lot of attention [Brázdil et al. 2014, 2015; Chakarov and Sankaranarayanan 2013, 2014; Chatterjee et al. 2016a,b, 2017; Claret et al. 2013; Cousot and Monerau 2012; Etessami et al. 2008; Etessami and Yannakakis 2005, 2015; Ferrer Fioriti and Hermanns 2015; Gehr et al. 2016; Kaminski et al. 2016; Katoen et al. 2010; Kattenbelt et al. 2009;

Olmedo et al. 2016; Sankaranarayanan et al. 2013]. Unfortunately, analyses of probabilistic programs have usually been standalone developments, and it is not immediately clear how different techniques relate.

This paper presents a framework, which we call *PMAF* (for Pre-Markov Algebra Framework), for designing, implementing, and proving the correctness of static analyses of probabilistic programs. We show how several analyses that may appear to be quite different, can be formulated—and generalized—using PMAF. Examples include Bayesian inference [Claret et al. 2013; Etessami and Yannakakis 2005, 2015], Markov decision problem with rewards [Puterman 1994], and probabilistic-invariant generation [Chakarov and Sankaranarayanan 2013; Chatterjee et al. 2016a; Katoen et al. 2010].

New constructs in probabilistic programs are of two kinds, to express *data randomness* (e.g., sampling) and *control-flow randomness* (e.g., probabilistic choice). To express both features, we introduce a new algebraic structure, called a *pre-Markov algebra*, which is equipped with operations corresponding to control-flow actions in probabilistic programs: *sequencing*, *conditional-choice*, *probabilistic-choice*, and *nondeterministic-choice*. PMAF is based on a new fixed-point semantics that models challenging features such as divergence, unstructured control-flow, nondeterminism, and continuous distributions. To establish correctness, we introduce *probabilistic abstractions* between two pre-Markov algebras that represent the concrete and abstract semantics.

Our work shows how, with suitable extensions, a blending of ideas from prior work on (i) static analysis of single-procedure probabilistic programs, and (ii) interprocedural dataflow analysis of standard (non-probabilistic) programs can be used to create a framework for interprocedural analysis of multi-procedure probabilistic programs. In particular,

- The semantics on which PMAF is based is an interpretation of the control-flow graphs (CFGs) for a program's procedures. One insight is to treat each CFG as a *hyper-graph* rather than a standard graph.
- The abstract semantics is formulated so that the analyzer can obtain *procedure summaries*.

Hyper-graphs contain *hyper-edges*, each of which consists of one source node and possibly several destination nodes. Conditional-branching, probabilistic-branching, and nondeterministic-branching statements are represented by hyper-edges. In ordinary CFGs, nodes can also have several successors; however, the operator applied at a confluence point q when analyzing a CFG is join ($\sqcup$), and the paths leading up to q are analyzed *independently*. For reasons discussed in §2.3, PMAF is based on a backward analysis, so the confluence points represent the program's branch points (i.e., for if-statements and while-loops). If the CFG is treated as a graph, join would be applied at each branch-node, and the subpaths from each successor would be analyzed independently. In contrast, when the CFG is treated as a hyper-graph, the operator applied at a probabilistic-choice node with probability p is $\lambda a.\lambda b.a \ {}_p\!\oplus b$—where ${}_p\!\oplus$ is not join, but an operator that weights the two successor paths by p and $1-p$. For instance, in Fig. 2(b), the hyper-edge $\langle v_0, \{v_1, v_5\}\rangle$ generates the inequality $\mathscr{A}[v_0] \sqsupseteq \mathscr{A}[v_1] \ {}_{0.75}\!\oplus \mathscr{A}[v_5]$, for some analysis $\mathscr{A}$. This approach allows the (hyper-)subpaths from the successors to be analyzed *jointly*.

To perform interprocedural analyses of probabilistic programs, we adopt a common practice from interprocedural analysis of standard non-probabilistic programs: the abstract domain is a *two-vocabulary* domain (each value represents an abstraction of a state transformer) rather than a *one-vocabulary* domain (each value represents an abstraction of a state). In the algebraic approach, an element in the algebra represents a two-vocabulary transformer. Elements can be "multiplied" by the algebra's formal multiplication operator, which is typically interpreted as (an abstraction of) the reversal of transformer composition. The transformer obtained for the set of hyper-paths from the entry of procedure P to the exit of P is the summary for P.

In the case of loops and recursive procedures, PMAF uses widening to ensure convergence. Here our approach is slightly non-standard: we found that for some instantiations of the framework, we could improve precision by using different widening operators for loops controlled by conditional, probabilistic, and nondeterministic branches.

The main advantage of PMAF is that instead of starting from scratch to create a new analysis, you only need to instantiate PMAF with the implementation of a new pre-Markov algebra. To establish soundness, you have to establish some well-defined algebraic properties, and can then rely on the soundness proof of the framework. To implement your analysis, you can rely on PMAF to perform sound interprocedural analysis, with respect to the abstraction that you provided. The PMAF implementation supplies common parts of different static analyses of probabilistic programs, e.g., efficient iteration strategies with widenings and interprocedural summarization. Moreover, any improvements made to the PMAF implementation immediately translate into improvements to *all of its instantiations*.

To evaluate PMAF, we created a prototype implementation, and reformulated two existing intraprocedural probabilistic-program analyses—the Bayesian-inference algorithm proposed by Claret et al. [Claret et al. 2013], and Markov decision problem with rewards [Puterman 1994]—to fit into PMAF: Reformulation involved changing from the one-vocabulary abstract domains proposed in the original papers to appropriate two-vocabulary abstract domains. We also developed a new program analysis: *linear expectation-invariant analysis* (LEIA). Linear expectation-invariants are equalities involving expected values of linear expressions over program variables.

A related approach to static analysis of probabilistic programs is *probabilistic abstract interpretation* (PAI) [Cousot and Monerau 2012; Monniaux 2000, 2001, 2003], which lifts standard program analysis to the probabilistic setting. PAI is both general and elegant, but the more concrete approach developed in our work on PMAF has a couple of advantages. First, PMAF is algebraic and provides a simple and well-defined interface for implementing new abstractions. We provide an actual implementation of PMAF that can be easily instantiated to specific abstract domains. Second, PMAF is based on a different semantic foundation, which follows the standard interpretation of non-deterministic probabilistic programs in domain theory [den Hartog and de Vink 1999; Jones 1989; Jones and Plotkin 1989; Mislove 2000; Mislove et al. 2004; Tix et al. 2009].

The concrete semantics of PAI isolates probabilistic choices from the non-probabilistic part of the semantics by interpreting programs as distributions $P : \Omega \to (D \to D)$, where Ω is a probability space and $D \to D$ is the space of non-probabilistic transformers. As a result, the PAI interpretation of the following non-deterministic program is that with probability $\frac{1}{2}$, we have a program that non-deterministically returns 1 or 2; with probability $\frac{1}{4}$, we have a program that returns 1; and with probability $\frac{1}{4}$, a program that returns 2.

$$\begin{array}{lll} \textbf{if} \star & \textbf{then} & \textbf{if prob}(\frac{1}{2})\ \textbf{then return}\ 1\ \textbf{else return}\ 2 \\ & \textbf{else} & \textbf{if prob}(\frac{1}{2})\ \textbf{then return}\ 1\ \textbf{else return}\ 2 \quad \textbf{fi} \end{array}$$

In contrast, the semantics used in PMAF resolves non-determinism on the outside, and thus the semantics of the program is that it returns 1 with probability $\frac{1}{2}$ and 2 with $\frac{1}{2}$. As a result, one can conclude that the expected return value r is 1.5. However, PAI—and every static analysis based on PAI—can only conclude $r \in \{1.25, 1.5, 1.75\}$.

Contributions. Our work makes five main contributions:

- We present a new denotational semantics for probabilistic programs, which is capable of expressing several nontrivial features of probabilistic-programming languages.
- We develop PMAF, an algebraic framework for static analyses of probabilistic programs. PMAF provides a novel approach to analyzing probabilistic programs with non-deterministic choice (§4.1 and §4.2) and general recursion (§4.3) (as well as continuous sampling and unstructured control-flow).
- We show that two previous intraprocedural probabilistic program analyses can be reformulated to fit into PMAF, thereby creating new interprocedural analyses for such previous work.

```
$b_1 \sim \mathsf{Bernoulli}(0.5)$;
$b_2 \sim \mathsf{Bernoulli}(0.5)$;
while $(\neg b_1 \wedge \neg b_2)$ do
  $b_1 \sim \mathsf{Bernoulli}(0.5)$;
  $b_2 \sim \mathsf{Bernoulli}(0.5)$
od
```
(a)

```
while prob($\frac{3}{4}$) do
  $z \sim \mathsf{Uniform}(0,2)$;
  if $\star$ then $x := x + z$
  else $y := y + z$
  fi
od
```
(b)

Fig. 1: (a) Boolean probabilistic program; (b) Arithmetic probabilistic program

- We develop a new program analysis, linear expectation-invariant analysis, by means of a suitable instantiation of PMAF. This analysis is more general than previous approaches to finding expectation invariants.
- We report on experiments with PMAF that show that the framework is easy to instantiate and works effectively. The experiments also show that linear expectation-invariant analysis can derive nontrivial invariants.

2 Overview

In this Section, we familiarize the reader with probabilistic programming, and briefly introduce two different static analyses of probabilistic programs: Bayesian inference and linear expectation invariant analysis. We then informally explain the main ideas behind our algebraic framework for analyzing probabilistic programs and show how it generalizes the aforementioned analyses.

2.1 Probabilistic Programming

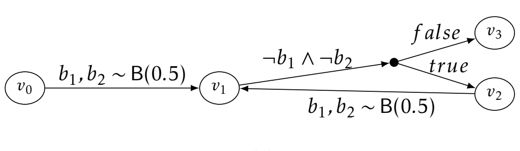
Probabilistic programs contain two sources of randomness: (i) *data randomness*, i.e., the ability to draw random values from distributions, and (ii) *control-flow randomness*, i.e., the ability to branch probabilistically. A variety of probabilistic programming languages and systems has been proposed [Carpenter et al. 2017; Goodman et al. 2008; Kok et al. 2007; Milch et al. 2005; Minka et al. 2014; Pfeffer 2005]. In this paper, our prototypical language is imperative.

We use the Boolean program in Fig. 1a to illustrate data randomness. In the program, b_1 and b_2 are two Boolean-valued variables. The *sampling statement* $x \sim \mathsf{Dist}(\bar{\theta})$ draws a value from a distribution Dist with a vector of parameters $\bar{\theta}$, and assigns it to the variable x, e.g., $b_1 \sim \mathsf{Bernoulli}(0.5)$ assigns to b_1 a random value drawn from a Bernoulli distribution with mean 0.5. Intuitively, the program tosses two fair Boolean-valued coins repeatedly, until one coin is *true*.

We introduce control-flow randomness through the arithmetic program in Fig. 1b. In the program, x, y, and z are real-valued variables. As in the previous example, we have sampling statements, and $\mathsf{Uniform}(l,r)$ represents a uniform distribution on the interval (l,r). The *probabilistic choice* **prob**(p) returns true with probability p and false with probability $1-p$. Moreover, the program also exhibits *nondeterminism*, as the symbol $\star$ stands for a *nondeterministic choice* that can behave like standard nondeterminism, as well as an arbitrary probabilistic choice [McIver and Morgan 2005, §6.6]. Intuitively, the program describes two players x and y playing a round-based game that ends with probability $\frac{1}{4}$ after each round. In each round, either player x or player y gains some reward that is uniformly distributed on $[0,2]$.

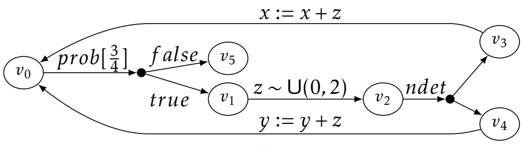
2.2 Two Static Analyses

Bayesian inference (BI). Probabilistic programs can be seen as descriptions of probability distributions [Carpenter et al. 2017; Goodman et al. 2008; Minka et al. 2014]. For a Boolean probabilistic pro-



$b_1, b_2 \sim \mathsf{B}(0.5)$
$\neg b_1 \wedge \neg b_2$
$false$
$true$
$b_1, b_2 \sim \mathsf{B}(0.5)$
v_0
v_1
v_2
v_3

(a)



(b)

Fig. 2: (a) Control-flow hyper-graph of the program in Fig. 1a. (b) Control-flow hyper-graph of the program in Fig. 1b.

gram, such as the one in Fig. 1a, *Bayesian-inference analysis* [Claret et al. 2013] calculates the distribution over variable valuations at the end of the program, conditioned on the program terminating. The inferred probability distribution is called the *posterior probability distribution*. The program in Fig. 1a specifies the posterior distribution over the variables (b_1, b_2) given by: $\mathbb{P}[b_1 = false, b_2 = false] = 0$, and $\mathbb{P}[b_1 = false, b_2 = true] = \mathbb{P}[b_1 = true, b_2 = false] = \mathbb{P}[b_1 = true, b_2 = true] = \frac{1}{3}$. This distribution also indicates that the program terminates *almost surely*, i.e., the probability that the program terminates is 1.[1]

Linear expectation invariant analysis (LEIA). Loop invariants are crucial to verification of imperative programs [Dijkstra 1976; Floyd 1967; Hoare 1969]. Although loop invariants for traditional programs are usually Boolean-valued expressions over program variables, real-valued invariants are needed to prove the correctness of probabilistic loops [Kozen 1981; McIver and Morgan 2005]. Such *expectation invariants* are usually defined as random variables—specified as expressions over program variables—with some desirable properties [Chakarov and Sankaranarayanan 2013, 2014; Katoen et al. 2010]. In this paper, we work with a more general kind of expectation invariant, defined as follows:

Definition 2.1. For a program P, $\mathbb{E}[\mathcal{E}_2] \bowtie \mathcal{E}_1$ is called an *expectation invariant* if $\mathcal{E}_1$ and $\mathcal{E}_2$ are real-valued expressions over P's program variables, $\bowtie$ is one of $\{=, <, >, \le, \ge\}$, and the following property holds: For any initial valuation of the program variables, the expected value of $\mathcal{E}_2$ in the final valuation (i.e., after the execution of P) is related to the value of $\mathcal{E}_1$ in the initial valuation by $\bowtie$.

We typically use variables with primes in $\mathcal{E}_2$ to denote the values in the final valuation. For example, for the program in Fig. 1b, $\mathbb{E}[x' + y'] = x + y + 3$, $\mathbb{E}[z'] = \frac{1}{4}z + \frac{3}{4}$, $\mathbb{E}[x'] \le x + 3$, $\mathbb{E}[x'] \ge x$, $\mathbb{E}[y'] \le y + 3$, and $\mathbb{E}[y'] \ge y$ are several linear expectation invariants, and our analysis can derive all of these automatically! The expectation invariant $\mathbb{E}[x' + y'] = x + y + 3$ indicates that the expected value of the total reward that the two players would gain is exactly 3.

2.3 The Algebraic Framework

This section explains the main ideas behind PMAF, which is general enough to encode the two analyses from §2.2.

Data Randomness vs. Control-Flow Randomness. Our first principle is to make an *explicit separation between data randomness and control-flow randomness*. This distinction is intended to make the framework more flexible for analysis designers by providing multiple ways to translate the constructs of their specific probabilistic programming language into the constructs of PMAF. Analysis designers may find it useful to use the control-flow-randomness construct directly (e.g., "**if prob**(0.3)..."), rather than simulating control-flow randomness by data randomness (e.g., "$p \sim \mathsf{Uniform}(0,1);$**if** $(p < 0.3)\ldots$"). For program analysis, such a simulation can lead to suboptimal results if the constructs used in the simulation require properties to be tracked that are outside the class of properties that a particular analysis's abstract domain is capable of tracking. For example, if an analysis domain only keeps track

[1] In general, we work with with *subprobability distributions*, where the probabilities add up to strictly less than 1. In the case of a program that diverges with probability $p > 0$, the posterior distribution is a subprobability distribution in which the probabilities of the states sum up to $1 - p$.

of expectations, then analysis of "$p \sim \mathsf{Uniform}(0,1)$" only indicates that $\mathbb{E}[p] = 0.5$, which does not provide enough information to establish that $\mathbb{P}[p < 0.3] = 0.3$ in the then-branch of "**if** $(p < 0.3)\ldots$". In contrast, when "**prob**$(0.3)\ldots$" is analyzed in the fragment with the explicit control-flow-randomness construct ("**if prob**$(0.3)\ldots$") the analyzer can directly assign the probabilities 0.3 and 0.7 to the outgoing branches, and use those probabilities to compute appropriate expectations in the respective branches.

We achieve the separation between data randomness and control-flow randomness by capturing the different types of randomness in the graphs that we use for representing programs. In contrast to traditional program analyses, which usually work on control-flow graphs (CFGs), we use *control-flow hyper-graphs* to model probabilistic programs. Hyper-graphs are directed graphs, each edge of which (i) has one source and possibly multiple destinations, and (ii) has an associated *control-flow action*—either *sequencing*, *conditional-choice*, *probabilistic-choice*, or *nondeterministic-choice*. A traditional CFG represents a collection of execution paths, while in probabilistic programs, paths are no longer independent, and the program specifies probability distributions over the paths. It is natural to treat a collection of paths as a whole and define distributions over the collections. These kinds of collections can be precisely formalized as *hyper-paths* made up of *hyper-edges* in hyper-graphs.

Fig. 2 shows the control-flow hyper-graphs of the two programs in Fig. 1. Every edge has an associated action, e.g., the control-flow actions $cond[\neg b_1 \wedge \neg b_2]$, $prob[\frac{3}{4}]$, and $ndet$ are conditional-choice, probabilistic-choice, and nondeterministic-choice actions. Data actions, like $x := x+z$ and $b_1 \sim \mathsf{Bernoulli}(0.5)$, also perform a trivial control-flow action to transfer control to their one destination node.

Just as the control-flow graph of a procedure typically has a single entry node and a single exit node, a procedure's control-flow hyper-graph also has a single entry node and a single exit node. In Fig. 2a, the entry and exit nodes are v_0 and v_3, respectively; in Fig. 2b, the entry and exit nodes are v_0 and v_5, respectively.

Backward Analysis. Traditional static analyses assign to a CFG node v either backward assertions—about the computations that can lead up to v—or forward assertions—about the computations that can continue from v [Cousot and Cousot 1977, 1979]. Backward assertions are computed via a forward analysis (in the same direction as CFG edges); forward assertions are computed via a backward analysis (counter to the flow of CFG edges).

Because we work with hyper-graphs rather than CFGs, from the perspective of a node v, 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.

The second principle of the framework is essentially dictated by this structural asymmetry: the framework *supports backward analyses that compute a particular kind of forward assertion*. In particular, the property to be computed for a node v in the control-flow hyper-graph for procedure P is (an abstraction of) a transformer that summarizes the transformation carried out by the hyper-graph fragment that extends from v to the exit node of P. It is possible to reason in the forward direction—i.e., about computations that lead up to v—but one would have to "break" hyper-paths into paths and "relocate" probabilities, which is more complicated than reasoning in the backward direction. The framework interprets an edge as a property transformer that computes properties of the edge's source node as a function of properties of the edge's destination node(s) and the edge's associated action. These property transformers propagate information in a *hypergraph-leaf-to-hypergraph-root* manner, which is natural in hyper-graph problems. For example, standard formulations of interprocedural dataflow analysis [Knoop and Steffen 1992; Lal et al. 2005; Müller-Olm and Seidl 2004; Sharir and Pnueli 1981] can be viewed as hyper-graph analyses, and propagation is performed in the leaf-to-root direction there as well.

Recall the Boolean program in Fig. 1a. Suppose that we want to perform BI to analyze $\mathbb{P}[b_1 = true, b_2 = true]$ in the posterior distribution. The property to be computed for a node will be a mapping from variable valuations to probabilities, where the probability reflects the chance that a given state will cause the program to terminate in the post-state $(b_1 = true, b_2 = true)$. For example, the property that we would hope to compute for node v_1 is the function $\lambda(b_1, b_2).[b_1 \wedge b_2] + [\neg b_1 \wedge \neg b_2] \cdot \frac{1}{3}$, where $[\varphi]$ is an *Iverson bracket*, which evaluates to 1 if φ is true, and 0 otherwise.

Two-Vocabulary Program Properties. In the example of BI above, we observe that the property transformation discussed above is not suitable for *interprocedural* analysis. Suppose that (i) we want analysis results to tell us something about $\mathbb{P}[b_1 = true, b_2 = true]$ in the posterior distribution of the main procedure, but (ii) to obtain the answer, the analysis must also analyze a call to some other procedure Q. In the main procedure, the analysis is driven by the posterior-probability query $\mathbb{P}[b_1 = true, b_2 = true]$; in general, however, Q will need to be analyzed with respect to some other posterior probability (obtained from the distribution of valuations at the point in main just after the call to Q). One might try to solve this issue by analyzing each procedure multiple times with different posterior probabilities. However, in an infinite state space, this approach is no longer feasible.

Following common practice in interprocedural static analysis of traditional programs, the third principle of the framework is to work with *two-vocabulary program properties*. The property sketched in the BI example above is actually *one-vocabulary*, i.e., the property assigned to a control-flow node only involves the state at that node. In contrast, a two-vocabulary property at node v (in the control-flow hyper-graph for procedure P) should describe the state transformation carried out by the hyper-graph fragment that extends from v to the exit node of P.

For instance, LEIA assigns to each control-flow node a conjunction of expectation invariants, which relate the state at the node to the state at the exit node; consequently, LEIA deals with two-vocabulary properties. In §5, we show that we can reformulate BI to manipulate two-vocabulary properties. As in interprocedural dataflow analysis [Cousot and Cousot 1978; Sharir and Pnueli 1981], procedure summaries are used to interpret procedure calls.

Separation of Concerns. Our fourth principle—which is common to most analysis frameworks—is *separation of concerns*, by which we mean

Provide a declarative interface for a client to specify the program properties to be tracked by a desired analysis, but leave it to the framework to furnish the analysis implementationby which the analysis is carried out.

We achieve this goal by adopting (and adapting) ideas from previous work on *algebraic program analysis* [Farzan and Kincaid 2015; Ramalingam 1996; Tarjan 1981]. Algebraic program analysis is based on the following idea:

> Any static analysis method performs reasoning in some space of program properties and property transformers; such property transformers should obey algebraic laws.

For instance, the data action **skip**, which does nothing, can be interpreted as the *identity* element in an algebra of program-property transformers.

Concretely, our fourth principle has three aspects:

1. For our intended domain of probabilistic programs, identify an appropriate set of algebraic laws that hold for useful sets of property transformers.
2. Define a specific algebra $\mathcal{A}$ for a program-analysis problem by defining a specific set of property transformers that obey the laws identified in item 1. Give translations from data actions and control-flow actions to such property transformers. (When such a translation is applied to a specific program, it sets up an equation system to be solved over $\mathcal{A}$.)
3. Develop a generic analysis algorithm that solves an equation system over any algebra that satisfies the laws identified in item 1.

Items 1 and 3 are tasks for us, the framework designers; they are the subjects of §3 and §4. Item 2 is a task for a client of the framework: examples are given in §5.

A client of the framework must furnish an *interpretation*—which consists of a *semantic algebra* and a *semantic function*—and a program. The semantic algebra consists of a *universe*, which defines the space

$$\begin{array}{ll}
\mathcal{S}(v_0) \sqsupseteq prob[\tfrac{3}{4}](\mathcal{S}(v_1), \mathcal{S}(v_5)) & \mathcal{S}(v_3) \sqsupseteq seq[x := x + z](\mathcal{S}(v_0)) \\
\mathcal{S}(v_1) \sqsupseteq seq[z \sim \mathsf{Uniform}(0,2)](\mathcal{S}(v_2)) & \mathcal{S}(v_4) \sqsupseteq seq[y := y + z](\mathcal{S}(v_0)) \\
\mathcal{S}(v_2) \sqsupseteq ndet(\mathcal{S}(v_3), \mathcal{S}(v_4)) & \mathcal{S}(v_5) \sqsupseteq \underline{1}
\end{array}$$

Fig. 3: The system of inequalities corresponding to Fig. 2b

of possible program-property transformers, and sequencing, conditional-choice, probabilistic-choice, and nondeterministic-choice operators, corresponding to control-flow actions. The semantic function is a mapping from data actions to the universe. (An interpretation is also called a *domain*.)

To address Item 3, our prototype implementation follows the standard *iterative* paradigm of static analysis [Cousot and Cousot 1977; Kildall 1973]: We first transform the control-flow hyper-graph into a system of inequalities, and then use a chaotic-iteration algorithm to compute a solution to it (e.g., [Bourdoncle 1993]), which repeatedly applies the interpretation until a fixed point is reached (possibly using widening to ensure convergence). For example, the control-flow hyper-graph in Fig. 2b can be transformed into the system shown in Fig. 3, where $\mathcal{S}(v) \in \mathcal{M}$ are elements in the semantic algebra; $\sqsubseteq$ is the approximation order on $\mathcal{M}$; $[\![\cdot]\!]$ is the semantic function, which maps data actions to $\mathcal{M}$; and $\underline{1}$ is the transformer associated with the exit node.

The soundness of the analysis (with respect to a concrete semantics) is proved by (i) establishing an approximation relation between the concrete domain and the abstract domain; (ii) showing that the abstract semantic function approximates the concrete one; and (iii) showing that the abstract operators (sequencing, conditional-choice, probabilistic-choice, and nondeterministic-choice) approximate the concrete ones.

For BI, we instantiate our framework to give lower bounds on posterior distributions, using with an interpretation in which state transformers are probability matrices (see §5.1). For LEIA, we design an interpretation using a Cartesian product of polyhedra (see §5.3). Once the functions of the interpretations are implemented, and a program is translated into the appropriate hyper-graph, the framework handles the rest of the work, namely, solving the equation system.

3 Probabilistic Programs

In this Section, we first review the concepts of *hyper-graphs* [Gallo et al. 1993] and introduce a probabilistic-program model based on them. Then we briefly sketch a new denotational semantics for our hyper-graph based imperative program model.

3.1 A Hyper-Graph Model of Probabilistic Programs

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 that 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 $\{H_i\}_{1 \le i \le n}$, where each procedure $H_i = \langle V_i, E_i, v_i^{\mathsf{entry}}, v_i^{\mathsf{exit}} \rangle$ is a control-flow hyper-graph in which each node except v_i^{exit} has exactly one outgoing hyper-edge. We assume that the nodes of each procedure are pairwise disjoint. To assign meanings to probabilistic programs modulo *data actions* $\mathcal{A}$ and *logical conditions* $\mathcal{L}$, we associate with each hyper-edge $e \in E = \bigcup_{1 \le i \le n} E_i$ a control-flow action $Ctrl(e)$, where

$$
\begin{aligned}
&\mathcal{A} ::= x := e \mid x \sim \mathscr{D} \mid \textbf{skip} \mid \textbf{observe}(\varphi)\\
&\varphi \in \mathcal{L} ::= \textbf{true} \mid \textbf{false} \mid e \bowtie u, \text{where } \bowtie\ \in \{=, \leq, \geq\} \mid \neg\varphi\\
&e, u \in \mathsf{Exp} ::= x \mid c, \text{where } c \in \mathbb{R} \mid e \bullet u, \text{where } \bullet \in \{+, -, \times, /\}\\
&x \in \mathsf{Var} ::= x \mid y \mid z \mid \cdots\\
&\mathscr{D} \in \mathsf{Dist} ::= \mathsf{Uniform}(e, u) \mid \mathsf{Gaussian}(e, u) \mid \cdots
\end{aligned}
$$

Fig. 4: Examples of data actions and logical conditions

Ctrl is

$$
\begin{aligned}
\mathit{Ctrl}\ ::=&\ \mathit{seq}[\text{act}] \text{ where act} \in \mathcal{A} \mid \mathit{call}[i] \text{ where } 1 \leq i \leq n\\
\mid&\ \mathit{cond}[\varphi] \text{ where } \varphi \in \mathcal{L} \mid \mathit{prob}[p] \text{ where } 0 \leq p \leq 1\\
\mid&\ \mathit{ndet}
\end{aligned}
$$

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

Fig. 2 shows two examples of hyper-graph–based probabilistic programs. See Fig. 4 for data actions $\mathcal{A}$ and logical conditions $\mathcal{L}$ that would be used for an arithmetic program like the one shown in Fig. 1b.

3.2 Background from Measure Theory

To define denotational semantics for probabilistic programs modulo data actions $\mathcal{A}$ and logical conditions $\mathcal{L}$, we review some standard definitions from measure theory [Billingsley 2012; Panangaden 1999].

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 another 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 the Borel space $\mathcal{B}(\mathbb{R}_{\geq 0})$ on nonnegative real numbers (the smallest σ-algebra containing all open intervals) is 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 $i \neq j$) 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 $\mu(X) \leq 1$. A *measure space* is a triple $\mathscr{M} = \langle X, \Sigma, \mu \rangle$ where μ is a measure on the measurable space $\langle X, \Sigma \rangle$. The *integral* of a Σ-measurable function f over the measurable 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 $\lambda A.[x \in A]$.

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 $\kappa : X_1 \times \Sigma_2 \to [0, 1]$ such that: (i) for each A in Σ_2, the function $\lambda x.\kappa(x, A)$ is Σ_1-measurable, and (ii) for each x in X_1, the function $\kappa_x \overset{\text{def}}{=} \lambda A.\kappa(x, A)$ is a distribution on $\langle X_2, \Sigma_2 \rangle$. We write the integral of a measurable function $f : \Sigma_2 \to [0, \infty]$ with respect to the distribution in (ii) as $\int f(y)\kappa(x, dy)$.

3.3 A Denotational Semantics

The next step is to define semantics based on the control-flow hyper-graphs. We use a denotational approach because it abstracts away how a program is evaluated and concentrates only on the effect of the program. This property makes it suitable as a starting point for static analysis, which is aimed at reasoning about program properties.

We develop a new semantics for probabilistic programming by combining Borgström et al.'s distribution-based semantics using the concept of kernels from measure theory [Borgström et al.

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

Fig. 5: Interpretation of actions and conditions

2016] and existing results on domain-theoretic probabilistic nondeterminism [den Hartog and de Vink 1999; Jones 1989; Jones and Plotkin 1989; Mislove 2000; Mislove et al. 2004; Tix et al. 2009]. This semantics can describe several nontrivial constructs, including continuous sampling, nondeterministic choice, and recursion.

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 act actions to *kernels* $\widehat{\mathrm{act}} : \Omega \times \mathcal{F} \to \mathbb{R}$. The intuition to keep in mind is that $\widehat{\mathrm{act}}(\omega, F)$ is the probability that the action, starting in state $\omega \in \Omega$, halts in a state that satisfies $F \in \mathcal{F}$ [Kozen 1985].[2]
- A mapping from logical conditions φ to measurable functions $\widehat{\varphi} : \Omega \to \{true, false\}$.

Example 3.3. For an arithmetic program with a finite set Var *of program variables,* Ω *is defined as* Var $\to \mathbb{R}$ *and* $\mathscr{P}$ *as the Borel space on* Ω*. Fig. 5 shows interpretation of the data actions and logical conditions in Fig. 4, where* $e(\omega)$ *evaluates expression* e *in state* ω*,* $(x \mapsto v)\omega$ *updates* x *in* ω *with* v*, and* $\mu_{\mathscr{D}} : \mathcal{B}(\mathbb{R}) \to [0, 1]$ *is the measure corresponding to the distribution* $\mathscr{D}$ *on reals. Note that the action* **observe**(φ) *performs* conditioning *on states that satisfy* φ*.*

While distributions can be seen as one-vocabulary specifications, kernels are indeed two-vocabulary transformers over program states. When there is no nondeterminism, we can assign a kernel to every control-flow node. We can also define control-flow actions on kernels. A sequence of actions $\mathrm{act}_1;\mathrm{act}_2$ with kernels κ_1 and κ_2, respectively, is modeled by their *composition*, denoted by $\kappa_1 \otimes \kappa_2$, which yields a new kernel defined as follows:[3]

$$\kappa_1 \otimes \kappa_2 \stackrel{\text{def}}{=} \lambda(x, A). \int \kappa_1(x, dy)\kappa_2(y, A). \tag{1}$$

The conditional-choice $\kappa_1 \,{}_{\varphi}\!\diamond\, \kappa_2$ is defined as a new kernel $\lambda(x, A).[\widehat{\varphi}(x)] \cdot \kappa_1(x, A) + [\neg\widehat{\varphi}(x)] \cdot \kappa_2(x, A)$. The probabilistic-choice $\kappa_1 \,{}_{p}\!\oplus\, \kappa_2$ is defined as a new kernel $\lambda(x, A).p \cdot \kappa_1(x, A) + (1 - p) \cdot \kappa_2(x, A)$.

Example 3.4. Consider the following program that models a variation on a geometric distribution.

```
$n := 0$;
while prob(0.9) do
  $n := n + 1$;
  if $n \geq 10$ then break else continue fi
od
```

[2] As explained by Kozen [Kozen 1985], for finite or countable Ω, $\widehat{\mathrm{act}}$ has a representation as a Markov transition matrix $M_{\widehat{\mathrm{act}}}$, in which each entry $M_{\widehat{\mathrm{act}}}(\omega, \omega')$ for a pair of states $\langle \omega, \omega' \rangle$ gives the probability that ω transitions to ω' under action act.

[3] For finite or countable Ω, and the matrix representation described in footnote 2, the integral in (1) degenerates to matrix multiplication [Kozen 1985].

Fig. 6 shows its control-flow hyper-graph. The assignment $n := n+1$ *is interpreted as a kernel* $\widehat{n := n+1} = \lambda(\omega,F).[(n \mapsto \omega(n)+1)\omega \in F]$*. The comparison* $n \geq 10$ *is interpreted as a measurable function* $\widehat{n \geq 10} = \lambda\omega.(\omega(n) \geq 10)$*. Let* K *stand for* 0.3486784401*; the semantics assigned to node* v_0 *is*

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

When nondeterminism comes into the picture, we need to associate each control-flow node with a *collection* of kernels. In other words, we need to consider *powerdomains* [Gunter et al. 1989] of kernels. We adopt Tix et al.'s constructions of probabilistic powerdomains [Tix et al. 2009], and extend them to work on kernels instead of distributions. We denote the set of feasible collections of kernels by $\mathbb{P}\Omega$, and the composition, conditional-choice, probabilistic-choice, and nondeterministic-choice operators on that by $\overline{\otimes}$, $\overline{_{\varphi}\Diamond}$, $\overline{_{p}\oplus}$, and $\overline{\Cup}$. $\mathbb{P}\Omega$ is also equipped with a partial order $\sqsubseteq$.

We reformulated distributions and kernels in a domain-theoretic way to adopt existing studies on powerdomains. We discuss the details of the construction of $\mathbb{P}\Omega$ in a companion paper [Wang et al. 2018]; the focus of this paper is static analysis and we will keep the domain-theoretic terminology to a minimum.

We adopted Hoare powerdomains and Smyth powerdomains [Abramsky and Jung 1994, §6.2] over kernels. Kernels are ordered pointwise, i.e., $\kappa_1 \leq \kappa_2$ if and only if for all ω and F, $\kappa_1(\omega,F) \leq \kappa_2(\omega,F)$. The zero kernel $\lambda(\omega,F).0$ is the bottom element of this order. Intuitively, the Hoare powerdomain is used for *partial* correctness, in which the order is set inclusion on the *lower* closures of the elements—because each downward-closed set contains a kernel that represents nontermination (i.e., the zero kernel), terminating and nonterminating executions cannot be distinguished. The Smyth powerdomain is used for *total* correctness: the order is reverse inclusion on the *upper* closures of the elements—nontermination is interpreted as the worst output, and the kernel that represents nontermination does not occur in an upward-closed set that represents the semantics of a terminating computation.

Given a probabilistic program $P = \{H_i\}_{1 \leq i \leq n}$, where $H_i = \langle V_i, E_i, v_i^{\mathsf{entry}}, v_i^{\mathsf{exit}} \rangle$, we want to define the semantics of each node v as a set of kernels that represent the effects from v to the exit node of the procedure that contains v. 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 $e = \langle v, \{u_1, \cdots, u_k\} \rangle \in E$, $\mathcal{S}\langle v \rangle = \widehat{Ctrl}(e)(\mathcal{S}\langle u_1 \rangle, \cdots, \mathcal{S}\langle u_k \rangle)$, and
- otherwise, $\mathcal{S}\langle v \rangle = \underline{1}_{\mathbb{P}}$.

The function $\widehat{\mathsf{act}}$ for the different kinds of control-flow actions is defined as follows:

$\widehat{seq[\mathsf{act}]}(S_1) \overset{\text{def}}{=} \{\widehat{\mathsf{act}}\}\overline{\otimes}S_1$	$\widehat{cond[\varphi]}(S_1,S_2) \overset{\text{def}}{=} S_1 \overline{_{\varphi}\Diamond} S_2$
$\widehat{call[i]}(S_1) \overset{\text{def}}{=} \mathcal{S}\langle v_i^{\mathsf{entry}} \rangle \overline{\otimes} S_1$	$\widehat{prob[p]}(S_1,S_2) \overset{\text{def}}{=} S_1 \overline{_{p}\oplus} S_2$
	$\widehat{ndet}(S_1,S_2) \overset{\text{def}}{=} S_1 \overline{\Cup} S_2$

Lemma 3.5. *The function* F_P *defined as*

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

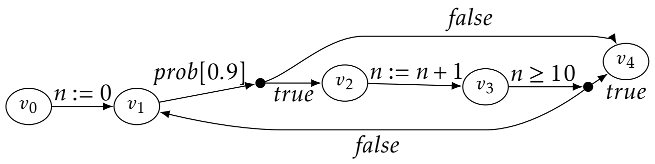


Fig. 6: Control-flow hyper-graph of the program in Ex. 3.4

is ω-continuous on $\langle V \to \mathbb{P}\Omega, \dot{\sqsubseteq}\rangle$, which is an ω-cpo with the least element $\lambda v.\underline{\perp}_{\mathbb{P}}$.

A dot over an operator denotes its application pointwise. By Kleene's fixed-point theorem, we have

Theorem 3.6. $\mathrm{lfp}^{\dot{\sqsubseteq}}_{\lambda v.\underline{\perp}_{\mathbb{P}}} F_P \stackrel{\text{def}}{=} \sup_{n\in\mathbb{N}}\{(F_P)^n(\lambda v.\underline{\perp}_{\mathbb{P}})\}$ *exists for all prob. programs P.*

Thus, the semantics of a node v is defined as $(\mathrm{lfp}^{\dot{\sqsubseteq}}_{\lambda v.\underline{\perp}_{\mathbb{P}}} F_P)(v)$.

4 Analysis Framework

To aid in creating abstractions of probabilistic programs, we first identify, in §4.1, some algebraic properties that underlie the mechanisms used in the semantics from §3.3. This algebra will aid our later definitions of abstractions in §4.2. We then discuss interprocedural analysis (in §4.3) and widening (§4.4). Proofs are included in appendix A.

4.1 An Algebraic Characterization of Fixpoint Semantics

In the denotational semantics, the concrete semantics is obtained by composing $\widehat{Ctrl(e)}$ operations along hyper-paths. Hence in the algebraic framework, the semantics of probabilistic programs is denoted by an *interpretation*, which consists of two parts: (i) a *semantic algebra*, which defines a set of possible program meanings, and which is equipped with sequencing, conditional-choice, probabilistic-choice, and nondeterministic-choice operators to compose these meanings, and (ii) a *semantic function*, which assigns a meaning to each basic program action.

The semantic algebras that we use—and the lattices used for abstract interpretation—are *Markov algebras* and *pre-Markov algebras*, respectively:

Definition 4.1 (Markov algebras). A *Markov algebra* (MA) over a set of logical conditions $\mathcal{L}$ is an 8-tuple $\mathcal{M} = \langle M, \sqsubseteq, \otimes, {}_{\varphi}\!\diamond, {}_{p}\!\oplus, \Cup, \underline{\perp}, \underline{1}\rangle$, where $\langle M, \sqsubseteq\rangle$ forms an ω-cpo with a least element $\underline{\perp}$; $\langle M, \otimes, \underline{1}\rangle$ 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 $\mathcal{L}$; ${}_{p}\!\oplus$ is a binary operator parametrized by $p \in [0,1]$; $\Cup$ is a binary operator that is idempotent, commutative, associative, and for all $a, b \in M$ and $p \in [0,1], \varphi \in \mathcal{L}$ we have $a \;{}_{p}\!\oplus\; b \le a \Cup b, a \;{}_{\varphi}\!\diamond\; b \le a \Cup b$ where $\le$ is the semilattice ordering induced by $\Cup$ (i.e., $a \le b$ if $a \Cup b = b$); and $\otimes$, ${}_{p}\!\oplus$, ${}_{\varphi}\!\diamond$, and $\Cup$ are ω-continuous.

Definition 4.2 (Pre-Markov algebras). A *pre-Markov algebra* (PMA) over a set of logical conditions $\mathcal{L}$ is an 8-tuple $\mathcal{M} = \langle M, \sqsubseteq, \otimes, {}_{\varphi}\!\diamond, {}_{p}\!\oplus, \Cup, \underline{\perp}, \underline{1}\rangle$, which is essentially an MA, *except* that $\langle M, \sqsubseteq\rangle$ forms a complete lattice; $\otimes$, ${}_{p}\!\oplus$, ${}_{\varphi}\!\diamond$, and $\Cup$ are only required to be monotone; and the following properties hold:

$$a \sqsubseteq a \;{}_{\varphi}\!\diamond\; a, \quad a \sqsubseteq a \;{}_{true}\!\diamond\; b, \qquad\qquad a \;{}_{\varphi}\!\diamond\; b = b \;{}_{\neg\varphi}\!\diamond\; a$$
$$a \sqsubseteq a \;{}_{p}\!\oplus\; a, \quad a \sqsubseteq a \;{}_{1}\!\oplus\; b\;\;, \qquad\qquad a \;{}_{p}\!\oplus\; b = b \;{}_{1-p}\!\oplus\; a$$
$$a \;{}_{\varphi}\!\diamond\; (b \;{}_{\psi}\!\diamond\; c) = (a \;{}_{\varphi'}\!\diamond\; b) \;{}_{\psi'}\!\diamond\; c \text{ where } \varphi = \varphi' \wedge \psi', \varphi \vee \psi = \psi'$$
$$a \;{}_{p}\!\oplus\; (b \;{}_{q}\!\oplus\; c) = (a \;{}_{p'}\!\oplus\; b) \;{}_{q'}\!\oplus\; c \text{ where } p = p'q', \overline{p} \cdot \overline{q} = \overline{q'}$$

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

Remark 4.3. These algebraic laws are not needed to prove soundness of the framework (stated in Thm. 4.7). These laws helped us when designing the abstract domains. Exploiting these algebraic laws to design better algorithms is an interesting direction for future work.

Lemma 4.4. *The denotational semantics in §3.3 is an MA $\mathcal{C} = \langle \mathbb{P}\Omega, \sqsubseteq_{\mathbb{P}}, \overline{\otimes}, \overline{{}_{\varphi}\!\diamond}, \overline{{}_{p}\!\oplus}, \overline{\Cup}, \underline{\perp}_{\mathbb{P}}, \underline{1}_{\mathbb{P}}\rangle$ (which we call the concrete domain for our framework).*

As is standard in abstract interpretation, the order on the algebra should represent an approximation order: $a \sqsubseteq b$ iff a is approximated by b (i.e., if a represents a more precise property than b).

Definition 4.5 (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 $\mathscr{I}[P]$ to be the interpretation of the probabilistic program. $\mathscr{I}[P]$ is then defined as the *least fixed point* of the function $F_P^\sharp$, which is defined as

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

where

$\widehat{seq_{\mathsf{act}}}^\sharp(a_1) \stackrel{\text{def}}{=} [\![\mathsf{act}]\!] \otimes a_1$	$\widehat{cond[\varphi]}^\sharp(a_1, a_2) \stackrel{\text{def}}{=} a_1 \, {}_{\varphi}\!\diamond \, a_2$
$\widehat{call[i]}^\sharp(a_1) \stackrel{\text{def}}{=} \mathcal{S}^\sharp(v_i^{\mathsf{entry}}) \otimes a_1$	$\widehat{prob[p]}^\sharp(a_1, a_2) \stackrel{\text{def}}{=} a_1 \, {}_{p}\!\oplus \, a_2$
	$\widehat{ndet}^\sharp(a_1, a_2) \stackrel{\text{def}}{=} a_1 \Cup a_2$

By the Knaster-Tarski theorem, we use the least fixed point of $F_P^\sharp$ to define the interpretation of a probabilistic program P as $\mathscr{I}[P] = \mathrm{lfp}^{\dot\sqsubseteq}_{\lambda v. \underline{\perp}} F_P^\sharp$. The interpretation of a control-flow node v is then defined as $\mathscr{I}[v] = \mathscr{I}[P](v)$.

4.2 Abstractions of Probabilistic Programs

Given an MA $\mathcal{C}$ and a PMA $\mathcal{A}$, a *probabilistic abstraction* is defined as follows:

Definition 4.6 (Probabilistic abstractions). A *probabilistic over-abstraction (or under-abstraction, resp.)* from a PMA $\mathcal{C}$ to a PMA $\mathcal{A}$ is a concretization mapping, $\gamma : \mathcal{A} \to \mathcal{C}$, such that

- $\underline{1}_{\mathcal{C}} \sqsubseteq_{\mathcal{C}} \gamma(\underline{1}_{\mathcal{A}})$ (or $\gamma(\underline{1}_{\mathcal{A}}) \sqsubseteq_{\mathcal{C}} \underline{1}_{\mathcal{C}}$, resp.),
- for all $Q_1, Q_2 \in \mathcal{A}$, $\gamma(Q_1) \otimes_{\mathcal{C}} \gamma(Q_2) \sqsubseteq_{\mathcal{C}} \gamma(Q_1 \otimes_{\mathcal{A}} Q_2)$ (or $\gamma(Q_1 \otimes_{\mathcal{A}} Q_2) \sqsubseteq_{\mathcal{C}} \gamma(Q_1) \otimes_{\mathcal{C}} \gamma(Q_2)$, resp.),
- for all $Q_1, Q_2 \in \mathcal{A}$, $\gamma(Q_1) \, {}_{\varphi}\!\diamond_{\mathcal{C}} \, \gamma(Q_2) \sqsubseteq_{\mathcal{C}} \gamma(Q_1 \, {}_{\varphi}\!\diamond_{\mathcal{A}} \, Q_2)$ (or $\gamma(Q_1 \, {}_{\varphi}\!\diamond_{\mathcal{A}} \, Q_2) \sqsubseteq_{\mathcal{C}} \gamma(Q_1) \, {}_{\varphi}\!\diamond_{\mathcal{C}} \, \gamma(Q_2)$, resp.),
- for all $Q_1, Q_2 \in \mathcal{A}$, $\gamma(Q_1) \, {}_{p}\!\oplus_{\mathcal{C}} \, \gamma(Q_2) \sqsubseteq_{\mathcal{C}} \gamma(Q_1 \, {}_{p}\!\oplus_{\mathcal{A}} \, Q_2)$ (or $\gamma(Q_1 \, {}_{p}\!\oplus_{\mathcal{A}} \, Q_2) \sqsubseteq_{\mathcal{C}} \gamma(Q_1) \, {}_{p}\!\oplus_{\mathcal{C}} \, \gamma(Q_2)$, resp.), and
- for all $Q_1, Q_2 \in \mathcal{A}$, $\gamma(Q_1) \Cup_{\mathcal{C}} \gamma(Q_2) \sqsubseteq_{\mathcal{C}} \gamma(Q_1 \Cup_{\mathcal{A}} Q_2)$ (or $\gamma(Q_1 \Cup_{\mathcal{A}} Q_2) \sqsubseteq_{\mathcal{C}} \gamma(Q_1) \Cup_{\mathcal{C}} \gamma(Q_2)$, resp.).

A probabilistic abstraction leads to a sound analyses:

Theorem 4.7. *Let $\mathscr{C}$ and $\mathscr{A}$ be interpretations over the MA $\mathcal{C}$ and the PMA $\mathcal{A}$; let γ be a probabilistic over-abstraction (or under-abstraction, resp.) from $\mathcal{C}$ to $\mathcal{A}$; and let P be an arbitrary probabilistic program. If for all basic actions* act, $[\![\mathsf{act}]\!]^{\mathscr{C}} \sqsubseteq_{\mathcal{C}} \gamma([\![\mathsf{act}]\!]^{\mathscr{A}})$ *(or $\gamma([\![\mathsf{act}]\!]^{\mathscr{A}}) \sqsubseteq_{\mathcal{C}} [\![\mathsf{act}]\!]^{\mathscr{C}}$, resp.), then we have $\mathscr{C}[P] \dot\sqsubseteq_{\mathcal{C}} \dot\gamma(\mathscr{A}[P])$ (or $\dot\gamma(\mathscr{A}[P]) \dot\sqsubseteq_{\mathcal{C}} \mathscr{C}[P]$, resp.).*

4.3 Interprocedural Analysis Algorithm

We are given a probabilistic program P and an interpretation $\mathscr{A} = \langle \mathcal{A}, [\![\cdot]\!]^{\mathscr{A}} \rangle$, where $\mathcal{A} = \langle M_{\mathcal{A}}, \sqsubseteq_{\mathcal{A}}, \otimes_{\mathcal{A}}, {}_{\varphi}\!\diamond_{\mathcal{A}}, {}_{p}\!\oplus_{\mathcal{A}}, \Cup_{\mathcal{A}}, \underline{\perp}_{\mathcal{A}}, \underline{1}_{\mathcal{A}} \rangle$ is a PMA and $[\![\cdot]\!]^{\mathscr{A}}$ is a semantic function. The goal is to compute (an overapproximation of) $\mathscr{A}[P] = \mathrm{lfp}^{\dot{\sqsubseteq}_{\mathcal{A}}}_{\lambda v.\underline{\perp}_{\mathcal{A}}} F^{\sharp}_{P}$. An equivalent way to define $\mathscr{A}[P]$ is to specify it as the least solution to a system of inequalities on $\{\mathscr{A}[v] \mid v \in V\}$ (where $e \in E$ in each case):

	e	$Ctrl(e)$
$\mathscr{A}[v] \sqsupseteq_{\mathcal{A}} [\![\mathrm{act}]\!]^{\mathscr{A}} \otimes_{\mathcal{A}} \mathscr{A}[u_1]$	$\langle v, \{u_1\}\rangle$	$seq[\mathrm{act}]$
$\mathscr{A}[v] \sqsupseteq_{\mathcal{A}} \mathscr{A}[u_1]\,{}_{\varphi}\diamond_{\mathcal{A}}\,\mathscr{A}[u_2]$	$\langle v, \{u_1, u_2\}\rangle$	$cond[\varphi]$
$\mathscr{A}[v] \sqsupseteq_{\mathcal{A}} \mathscr{A}[u_1]\,{}_{p}\oplus_{\mathcal{A}}\,\mathscr{A}[u_2]$	$\langle v, \{u_1, u_2\}\rangle$	$prob[p]$
$\mathscr{A}[v] \sqsupseteq_{\mathcal{A}} \mathscr{A}[u_1] \Cup_{\mathcal{A}} \mathscr{A}[u_2]$	$\langle v, \{u_1, u_2\}\rangle$	$ndet$
$\mathscr{A}[v] \sqsupseteq_{\mathcal{A}} \mathscr{A}[v_i^{\mathsf{entry}}] \otimes_{\mathcal{A}} \mathscr{A}[u_1]$	$\langle v, \{u_1\}\rangle$	$call[i]$
$\mathscr{A}[v] \sqsupseteq_{\mathcal{A}} \underline{1}_{\mathcal{A}}$	if $v = v_i^{\mathsf{exit}}$	

Note that in line 5 a call is treated as a hyper-edge with the action $\lambda(entry, succ).entry \otimes_{\mathcal{A}} succ$. There is no explicit return edge to match a call (as in many multi-procedure program representations, e.g., [Reps et al. 1995]); instead, each exit node is initialized with the constant $\underline{1}_{\mathcal{A}}$ (line 6).

We mainly use known techniques from previous work on interprocedural dataflow analysis, with some adaptations to our setting, which uses hyper-graphs instead of ordinary graphs (i.e., CFGs).[4] The analysis direction is backward, and the algorithm is similar to methods for computing summary edges in demand interprocedural-dataflow-analysis algorithms ([Horwitz et al. 1995, Fig. 4], [Sagiv et al. 1996, Fig. 10]). The algorithm uses a standard chaotic-iteration strategy (except that propagation is performed along hyper-edges instead of edges); it uses a fair iteration strategy for selecting the next edge to consider.

4.4 Widening

Widening is a general technique in static analysis to ensure and speed up convergence [Cousot 1981; Cousot and Cousot 1978]. To choose the nodes at which widening is to be applied, we treat the hyper-graph as a graph—i.e., each hyper-edge (including calls) contributes one or two ordinary edges. More precisely, we construct a *dependence graph* $G(H) = \langle N, A\rangle$ from hyper-graph $H = \{\langle V_i, E_i, v_i^{\mathsf{entry}}, v_i^{\mathsf{exit}}\rangle\}_{1\le i\le n}$ by defining $N \overset{\mathrm{def}}{=} \bigcup_{1\le i\le n} V_i$, and

$$A \overset{\mathrm{def}}{=} \{\langle u, v\rangle \mid \exists e \in E.(v = src(e) \wedge u \in Dst(e))\} \cup \{\langle v_i^{\mathsf{entry}}, v\rangle \mid \exists e \in E.(v = src(e) \wedge Ctrl(e) = call[i])\}. \quad (2)$$

We then compute a set W of widening points for $G(H)$ via the algorithm of Bourdoncle [Bourdoncle 1993, Fig. 4]. Because of the second set-former in (2), W contains widening points that cut each cycle caused by recursion.

While traditional programs exhibit only one sort of choice operator, probabilistic programs can have three different kinds of choice operators, and hence loops can exhibit three different kinds of behavior. We found that if we used the same widening operator for all widening nodes, there could be a substantial loss in precision. Thus, we equip the framework with three separate widening operators:

[4]As mentined in §2.3, standard formulations of interprocedural dataflow analysis [Knoop and Steffen 1992; Lal et al. 2005; Müller-Olm and Seidl 2004; Sharir and Pnueli 1981] can be viewed as hyper-graph analyses. In that setting, one deals with hyper-graphs with consituent control-flow graphs. With PMAF, because each procedure is represented as a hyper-graph, one has hyper-graphs of constituent hyper-graphs. Fortunately, each procedure's hyper-graph is a *single-entry/single-exit* hyper-graph, so the basic ideas and algorithms from standard interprocedural dataflow analysis carry over to PMAF.

$\triangledown_c$, $\triangledown_p$, and $\triangledown_n$. Let $v \in W$ be the source of edge $e \in E$. Then the inequalities become

	e	$Ctrl(e)$
$\mathscr{A}[v] \sqsupseteq_{\mathcal{A}} \mathscr{A}[v] \triangledown_n (\llbracket \mathsf{act} \rrbracket^{\mathscr{A}} \otimes_{\mathcal{A}} \mathscr{A}[u_1])$	$\langle v, \{u_1\} \rangle$	$seq[\mathsf{act}]$
$\mathscr{A}[v] \sqsupseteq_{\mathcal{A}} \mathscr{A}[v] \triangledown_c (\mathscr{A}[u_1] \,{}_{\varphi}\!\diamond_{\mathcal{A}}\, \mathscr{A}[u_2])$	$\langle v, \{u_1, u_2\} \rangle$	$cond[\varphi]$
$\mathscr{A}[v] \sqsupseteq_{\mathcal{A}} \mathscr{A}[v] \triangledown_p (\mathscr{A}[u_1] \,{}_{p}\!\oplus_{\mathcal{A}}\, \mathscr{A}[u_2])$	$\langle v, \{u_1, u_2\} \rangle$	$prob[p]$
$\mathscr{A}[v] \sqsupseteq_{\mathcal{A}} \mathscr{A}[v] \triangledown_n (\mathscr{A}[u_1] \Cup_{\mathcal{A}} \mathscr{A}[u_2])$	$\langle v, \{u_1, u_2\} \rangle$	$ndet$
$\mathscr{A}[v] \sqsupseteq_{\mathcal{A}} \mathscr{A}[v] \triangledown_n (\mathscr{A}[v_i^{\mathsf{entry}}] \otimes_{\mathcal{A}} \mathscr{A}[u_1])$	$\langle v, \{u_1\} \rangle$	$call[i]$

Observation 4.8. Recall from Defn. 3.2 that in a probabilistic program each non-exit node has exactly one outgoing hyper-edge. In each right-hand side above, the second argument to the widening operator re-evaluates the action of the (one outgoing) hyper-edge. Consequently, during an analysis, we have the invariant that whenever a widening operation $a \triangledown b$ is performed, the property $a \sqsubseteq_{\mathcal{A}} b$ holds.

The safety properties for the three widening operators are adaptations of the standard stabilization condition: For every pair of ascending chains $\{a_k\}_{k \in \mathbb{N}}$ and $\{b_k\}_{k \in \mathbb{N}}$,

- the chain $\{c_k\}_{k \in \mathbb{N}}$ defined by $c_0 = a_0 \,{}_{\varphi}\!\diamond_{\mathcal{A}}\, b_0$ and $c_{k+1} = c_k \triangledown_c (a_{k+1} \,{}_{\varphi}\!\diamond_{\mathcal{A}}\, b_{k+1})$ is eventually stable;
- the chain $\{c_k\}_{k \in \mathbb{N}}$ defined by $c_0 = a_0 \,{}_{p}\!\oplus_{\mathcal{A}}\, b_0$ and $c_{k+1} = c_k \triangledown_p (a_{k+1} \,{}_{p}\!\oplus_{\mathcal{A}}\, b_{k+1})$ is eventually stable; and
- the chain $\{c_k\}_{k \in \mathbb{N}}$ defined by $c_0 = a_0 \Cup_{\mathcal{A}} b_0$ and $c_{k+1} = c_k \triangledown_n (a_{k+1} \Cup_{\mathcal{A}} b_{k+1})$ is eventually stable.

5 Instantiations

In this Section, we instantiate the framework to derive three important analyses: Bayesian inference (BI) (§5.1), computing rewards in Markov decision processes (§5.2), and linear expectation-invariant analysis (LEIA) (§5.3). Proofs are included in appendix A.

5.1 Bayesian Inference

Claret et al. [Claret et al. 2013] proposed a technique to perform Bayesian inference on Boolean programs using dataflow analysis. They use a forward analysis to compute the posterior distribution of a single-procedure, well-structured, probabilistic program. Their analysis is similar to an intraprocedural dataflow analysis: they use discrete joint-probability distributions as dataflow facts, merge these facts at join points, and compute fixpoints in the presence of loops. Let Var be the set of program variables; the set of program states is $\Omega = \mathsf{Var} \to \mathbb{B}$. Note that Ω is isomorphic to $\mathbb{B}^{|\mathsf{Var}|}$, and consequently, a distribution can be represented by a vector of length $2^{|\mathsf{Var}|}$ of reals in $\mathbb{R}_{[0,1]}$. (Their implementation uses Algebraic Decision Diagrams [Bahar et al. 1997] to represent distributions compactly.)

The algorithm by Claret et al. is defined inductively on the structure of programs [Claret et al. 2013, Alg. 2]—for example, the output distribution of $x \sim \mathsf{Bernoulli}(r)$ from an input distribution μ, denoted by $\textsc{Post}(\mu, x \sim \mathsf{Bernoulli}(r))$, is computed as $\lambda\sigma'.\Big(r \cdot \sum_{\{\sigma \mid \sigma' = \sigma[x \leftarrow true]\}} \mu(\sigma) + (1-r) \cdot \sum_{\{\sigma \mid \sigma' = \sigma[x \leftarrow false]\}} \mu(\sigma)\Big)$.

We have used PMAF to extend their work in two dimensions, creating (i) an *interprocedural* version of Bayesian inference with (ii) *nondeterminism*. Because of nondeterminism, for a given input state the posterior distribution is not unique; consequently, our goal is to compute procedure summaries that gives *lower bounds* on posterior distributions.

To reformulate the domain in the two-vocabulary setting needed for computing procedure summaries, we introduce Var', primed versions of the variables in Var. Var and Var' denote the variables in the pre-state and post-state of a state transformer. A distribution transformer (and therefore a procedure summary) is a matrix of size $2^{|\mathsf{Var}|} \times 2^{|\mathsf{Var}'|}$ of reals in $\mathbb{R}_{[0,1]}$. We define a PMA $\mathcal{B} = \langle M_{\mathcal{B}}, \sqsubseteq_{\mathcal{B}}, \otimes_{\mathcal{B}}, {}_{\varphi}\diamond_{\mathcal{B}}, {}_{p}\oplus_{\mathcal{B}}, \Cup_{\mathcal{B}}, \bot_{\mathcal{B}}, \underline{1}_{\mathcal{B}} \rangle$ as follows:

$M_{\mathcal{B}} \overset{\text{def}}{=} 2^{\mathsf{\|Var\|}} \times 2^{\mathsf{\|Var'\|}} \to \mathbb{R}_{[0,1]}$	
$a \sqsubseteq_{\mathcal{B}} b \overset{\text{def}}{=} a \dot{\le} b$	$a \Cup_{\mathcal{B}} b \overset{\text{def}}{=} \dot{\min}(a,b)$
$a \otimes_{\mathcal{B}} b \overset{\text{def}}{=} a \times b$	$\bot_{\mathcal{B}} \overset{\text{def}}{=} \lambda(s,t).0$
$a\, {}_{p}\oplus_{\mathcal{B}}\, b \overset{\text{def}}{=} p \cdot a + (1-p) \cdot b$	$\underline{1}_{\mathcal{B}} \overset{\text{def}}{=} \lambda(s,t).[s=t])$
$a\, {}_{\varphi}\diamond_{\mathcal{B}}\, b \overset{\text{def}}{=} \lambda(s,t).\textbf{if } \widehat{\varphi}(s) \textbf{ then } a(s,t) \textbf{ else } b(s,t)$	

The use of pointwise min in the definition of $a \Cup_{\mathcal{B}} b$ causes the analysis to compute procedure summaries that provide lower bounds on the posterior distributions.

Let $\mathscr{B} = \langle \mathcal{B}, [\![\cdot]\!]^{\mathscr{B}} \rangle$ be the interpretation for Bayesian inference. We define the semantic function as $[\![x := \mathcal{E}]\!]^{\mathscr{C}} = \lambda(s,A).[s[x \leftarrow \mathcal{E}(s)] \in A]$ and $[\![x := \mathcal{E}]\!]^{\mathscr{B}} = \lambda(s,t).[s[x \leftarrow \mathcal{E}(s)] = t]$, as well as $[\![x \sim \mathsf{Bernoulli}(p)]\!]^{\mathscr{C}} = \lambda(s,A).p \cdot [s[x \leftarrow true] \in A] + (1-p) \cdot [s[x \leftarrow false] \in A]$ and $[\![x \sim \mathsf{Bernoulli}(p)]\!]^{\mathscr{B}} = \lambda(s,t).p \cdot [s[x \leftarrow true] = t] + (1-p) \cdot [s[x \leftarrow false] = t])$.

We define the concretization mapping $\gamma_{\mathcal{B}} : M_{\mathcal{B}} \to \mathbb{P}\Omega$ as $\gamma_{\mathcal{B}}(a) = \langle\!\langle \{\kappa \mid \forall s,s'.\kappa(s,\{s'\}) \ge a(s,s')\} \rangle\!\rangle$ where $\langle\!\langle C \rangle\!\rangle$ denotes the smallest element in $\mathbb{P}\Omega$ such that contains C.

Theorem 5.1. *$\gamma_{\mathcal{B}}$ is a prob. under-abstraction from $\mathcal{C}$ to $\mathcal{B}$.*

We do not define widening operators for BI, because $\gamma_{\mathcal{B}}$ is an under-abstraction and our algorithm starts from the bottom element in the abstract domain, the intermediate result at any iteration is a sound answer.

5.2 Markov Decision Process with Rewards

Analyses of finite-state Markov decision processes were originally developed in the fields of operational research and finance mathematics [Puterman 1994]. Originally, Markov decision processes were defined as finite-state machines with actions that exhibit probabilistic transitions. In this paper, we use a slightly different formalization, using hyper-graphs.

Definition 5.2 (Markov decision process). A *Markov decision process* (MDP) is a hyper-graph $H = \langle V, E, v^{\mathsf{entry}}, v^{\mathsf{exit}} \rangle$, where every node except v^{exit} has exactly one outgoing hyper-edge; each hyper-edge with just a single destination has an associated *reward*, $seq[\textbf{reward}(r)]$, where r is a positive real number; and each hyper-edge with two destinations has either $prob[p]$, where $0 \le p \le 1$, or $ndet$. Note that MDPs are a specialization of single-procedure probabilistic programs without conditional-choice.

We can also treat the hyper-graph as a graph: each hyper-edge contributes one or two graph edges. A path through the graph has a *reward*, which is the sum of the rewards that label the edges of the path. (Edges from hyper-edges with the actions $prob[p]$ or $ndet$ are considered to have reward 0.) The analysis problem that we wish to solve is to determine, for each node v, the greatest expected reward that one can gain by executing the program from v.

It is natural to extend MDPs with procedure calls and multiple procedures, to obtain *recursive Markov decision processes*. The set of program states is defined to be the set of nonnegative real numbers: $\Omega = [0,\infty]$. To address the maximum-expected-reward problem for a recursive Markov decision process, we define a PMA $\mathcal{R} = \langle M_\mathcal{R}, \sqsubseteq_\mathcal{R}, \otimes_\mathcal{R}, {}_\varphi\diamond_\mathcal{R}, {}_p\oplus_\mathcal{R}, \Cup_\mathcal{R}, \bot_\mathcal{R}, \underline{1}_\mathcal{R}\rangle$ as follows:

$M_\mathcal{R} \overset{\text{def}}{=} [0,\infty]$	${}_\varphi\diamond_\mathcal{R} \overset{\text{def}}{=} \max$	$\bot_\mathcal{R} \overset{\text{def}}{=} 0$
$\sqsubseteq_\mathcal{R} \overset{\text{def}}{=} \le$	$a\ {}_p\oplus_\mathcal{R}\ b \overset{\text{def}}{=} p\cdot a + (1-p)\cdot b$	$\underline{1}_\mathcal{R} \overset{\text{def}}{=} 0$
$\otimes_\mathcal{R} \overset{\text{def}}{=} +$	$\Cup_\mathcal{R} \overset{\text{def}}{=} \max$	

Let $\mathscr{R} = \langle \mathcal{R}, [\![\cdot]\!]^\mathscr{R}\rangle$ be the interpretation for a Markov decision process with rewards. We define the semantic function as $[\![\mathbf{reward}(r)]\!]^\mathscr{C} = \lambda(s,A).[s+r\in A]$ and $[\![\mathbf{reward}(r)]\!]^\mathscr{R} = r$.

We define the concretization mapping $\gamma_\mathcal{R} : M_\mathcal{R} \to \mathbb{P}[0,\infty]$ as follows: $\gamma_\mathcal{R}(a) = \langle\!\langle \{\kappa \mid \forall s. \int y\cdot\kappa(s,dy) \le s+a\}\rangle\!\rangle$.

Theorem 5.3. *$\gamma_\mathcal{R}$ is a prob. over-abstraction from $\mathcal{C}$ to $\mathcal{R}$.*

We use a trivial widening in this analysis: if after some fixed number of iterations the analysis does not converge, it returns ∞ as the result.

5.3 Linear Expectation-Invariant Analysis

Several examples of expectation invariants obtained via linear expectation-invariant analysis (LEIA) were given in §2.2. This section gives details of the abstract domain for LEIA.

We make use of an existing abstract domain, namely, the domain of *convex polyhedra* [Cousot and Halbwachs 1978]. Elements of the polyhedral domain are defined by linear-inequality and linear-equality constraints among program variables. For LEIA, we use two-vocabulary polyhedra over nonnegative program variables. Let $x = (x_1,\cdots,x_n)^T$ be a column vector of nonnegative program variables and $x' = (x'_1,\cdots,x'_n)^T$ be a column vector of the "primed" versions of corresponding program variables. A polyhedron $P \subseteq \mathbb{R}^{2n}_{\ge 0}$ captures linear-inequality constraints among x and x', which can be interpreted as a relation between pre-state and post-state variable valuations.

A polyhedron $P = \{(x'^T x^T)^T \in \mathbb{R}^{2n}_{\ge 0} \mid A'x' + Ax \le b \wedge D'x' + Dx = e\}$, can be encoded as the intersection of a finite number of closed half spaces and a finite number of subspaces, where A', A, D', D are matrices and b, e are vectors. The associated *constraint set* is defined as $\mathcal{C}_P = \{A'x' + Ax \le b, D'x' + Dx = e\}$. Let $\mathcal{P}$ be the set of polyhedra; $\mathcal{P}$ is equipped with meet, join, renaming, forgetting, and comparison operations.

LEIA uses *expectation polyhedra*. They are actually the same as polyhedra, except that the two vocabularies are $x = (x_1,\cdots,x_n)^T$ and $\mathbb{E}[x'] = (\mathbb{E}[x'_1],\cdots,\mathbb{E}[x'_n])^T$. An expectation polyhedron represents a constraint set of the form

$$\{A'\mathbb{E}[x'] + Ax \le b, D'\mathbb{E}[x'] + Dx = e\}. \tag{3}$$

Because of the linearity of the expectation operator $\mathbb{E}$, an equivalent way to express (3) is as follows:

$$\{\mathbb{E}[A'x'] + Ax \le b, \mathbb{E}[D'x'] + Dx = e\}.$$

Let $\mathcal{EP}$ be the set of expectation polyhedra. $\mathcal{EP}$ is equipped with the same set of operations as $\mathcal{P}$.

We define the state space to be $\Omega = \mathbb{R}^n_{\ge 0}$. We then define a PMA $\mathcal{I}$ with a universe $M_\mathcal{I} \overset{\text{def}}{=} \mathcal{P} \times \mathcal{EP}$. An element $(P, EP) \in \mathcal{I}$ consists of (i) a set of standard constraints $P \in \mathcal{P}$, and (ii) a set of expectation constraints $EP \in \mathcal{EP}$, such that $\mathbf{0} \sqcup P[\mathbb{E}[x']/x'] \sqsupseteq EP$ holds, where $\mathbf{0} \overset{\text{def}}{=} \bigwedge_{i=1}^n (\mathbb{E}[x'_i] = 0)$. The latter property means that, if necessary, we can always "rebuild" a pessimistic $\mathcal{EP}$ component from the $\mathcal{P}$ component as $\mathbf{0} \sqcup P[\mathbb{E}[x']/x']$.[5]

[5]The intuition is that P represents a convex overapproximation to some desired set of points; the expected value has to lie somewhere inside $\vec{0} \sqcup P$, where "$\vec{0} \sqcup \ldots$" is needed to account for subprobability distributions. For instance, for a nonnegative interval $[lo, hi]$, we must have $expected \in ([0,0] \sqcup [lo, hi])$; i.e., $0 \le expected \le hi$.

We define the concretization mapping $\gamma_{\mathcal{I}}$ as follows:

$$\gamma_{\mathcal{I}}(P, EP) = \langle\!\langle \left\{ \kappa \mid \forall s. \kappa\left(s, \left\{ s' \mid \left[\begin{array}{c} s' \\ s \end{array} \right] \models \neg P \right\}\right) = 0 \ \wedge \left[\begin{array}{c} \int s' \kappa(s, ds') \\ s \end{array} \right] \models EP \right\} \rangle\!\rangle.$$

Comparison. The comparison operation on ordinary polyhedra can be defined as standard set inclusion. For expectation polyhedra, taking into account subprobability distributions, we define $EP_1 \sqsubseteq EP_2$ to be $\mathbf{0} \sqcup EP_1 \subseteq \mathbf{0} \sqcup EP_2$, so that any element inside or below EP_1 should also be inside or below EP_2. Consequently, we define $(P_1, EP_1) \sqsubseteq_{\mathcal{I}} (P_2, EP_2) \stackrel{\text{def}}{=} P_1 \subseteq P_2 \wedge \mathbf{0} \sqcup EP_1 \subseteq \mathbf{0} \sqcup EP_2$.

Composition. For ordinary polyhedra, the composition of P_1 and P_2 can be defined as

$$(\exists x''. \mathcal{C}_{P_1}[x''/x'] \wedge \mathcal{C}_{P_2}[x''/x]) \Rightarrow \mathcal{C}_{P_1 \otimes P_2},$$

where we introduce an intermediate vocabulary $x'' = (x_1'', \cdots, x_n'')^T$, and use it to connect P_1 and P_2. Consequently, we define $P_1 \otimes P_2$ to be $\exists x''. \mathcal{C}_{P_1}[x''/x'] \wedge \mathcal{C}_{P_2}[x''/x]$. Operationally, composition involves first introducing a new vocabulary; renaming the variables properly; performing a meet, and finally forgetting the intermediate vocabulary.

Somewhat surprisingly, because of the *tower property* in probability theory, exactly the same steps can be used to compose expectation polyhedra. Informally, the tower property means that $\mathbb{E}[X] = \mathbb{E}[\mathbb{E}[X \mid Y]]$, where X and Y are two random variables, and $\mathbb{E}[X \mid Y]$ is a conditional expectation. For instance, suppose that EP_1 and EP_2 are defined by the constraint sets $\{\mathbb{E}(x') = x + 2\}$ and $\{\mathbb{E}(x') = 7x\}$, respectively. Following the renaming recipe above, we have $\mathbb{E}(x'') = x + 2$ and $\mathbb{E}(x' \mid x'') = 7x''$. By the tower property, we have $\mathbb{E}(x') = \mathbb{E}(\mathbb{E}(x' \mid x'')) = \mathbb{E}(7x'') = 7\mathbb{E}(x'') = 7x + 14$. Operationally, the tower property allows us to compose linear expectation invariants, and eliminate the intermediate vocabulary x''. Consequently, we define

$$(P_1, EP_1) \otimes_{\mathcal{I}} (P_2, EP_2) \stackrel{\text{def}}{=} (P_1 \otimes P_2, EP_1 \otimes EP_2).$$

Conditional-choice. For the ordinary-polyhedron component, a conditional-choice ${}_{\varphi}\Diamond$ is performed by first meeting each operand with the logical constraint φ, and then joining the results. However, for the expectation-polyhedron component, conditioning can split the probability space in almost arbitrary ways. Consequently, the constraints on post-state expectations as a function of pre-state valuations are not necessarily true after conditioning. Thus, we define

$$\begin{aligned}(P_1, EP_1) \,{}_{\varphi}\Diamond_{\mathcal{I}}\, (P_2, EP_2) \stackrel{\text{def}}{=} \ & \textbf{let } P = (\{\varphi\} \sqcap P_1) \sqcup (\{\neg\varphi\} \sqcap P_2) \\ & \textbf{in } (P, (EP_1 \sqcup EP_2) \sqcap (\mathbf{0} \sqcup P[\mathbb{E}[x']/x'])).\end{aligned}$$

The $\sqcap$ in the second component is performed to maintain the invariant that $\mathbf{0} \sqcup P[\mathbb{E}[x']/x'] \sqsupseteq$ the second component.

Probabilistic-choice. For the ordinary-polyhedron component, we merely join the components of the two operands. For the expectation-polyhedron component, we introduce two more vocabularies and have

$$(\exists x'', x'''. \mathcal{C}_{EP_1}[x''/\mathbb{E}[x']] \wedge \mathcal{C}_{EP_2}[x'''/\mathbb{E}[x']] \wedge \bigwedge_{i=1}^{n} \mathbb{E}[x_i'] = p \cdot x_i'' + (1-p) \cdot x_i''') \Rightarrow \mathcal{C}_{EP_1 \,{}_p\oplus EP_2}.$$

Consequently, we define $EP_1 \,{}_p\oplus EP_2$ to be

$$\exists x'', x'''. \left(\mathcal{C}_{EP_1}[x''/\mathbb{E}[x']] \wedge \mathcal{C}_{EP_2}[x'''/\mathbb{E}[x']] \wedge \bigwedge_{i=1}^{n} \mathbb{E}[x_i'] = p \cdot x_i'' + (1-p) \cdot x_i''' \right),$$

and $(P_1, EP_1)\ {}_p{\oplus}_{\mathcal{I}}\ (P_2, EP_2) \stackrel{\text{def}}{=} (P_1 \sqcup P_2, EP_1\ {}_p{\oplus}\ EP_2)$.

Nondeterministic-choice. The nondeterministic-choice operations on both ordinary polyhedra and expectation polyhedra can be defined as join. Hence, we define $(P_1, EP_1) \Cup_{\mathcal{I}} (P_2, EP_2) \stackrel{\text{def}}{=} (P_1 \sqcup P_2, EP_1 \sqcup EP_2)$.

Bottom and Unit Element. We define $\bot_{\mathcal{I}} \stackrel{\text{def}}{=} (false, \mathbf{0})$, and $\underline{1}_{\mathcal{I}} \stackrel{\text{def}}{=} (\{x_i' = x_i \mid 1 \le i \le n\}, \{\mathbb{E}[x_i'] = x_i \mid 1 \le i \le n\})$.

Semantic Function. Some examples of the semantic mapping $[\![\cdot]\!]^{\mathscr{I}}$ are as follows, where $\min(\mathscr{D})$ and $\max(\mathscr{D})$ represents the interval of the support of a distribution $\mathscr{D}$, while $\text{mean}(\mathscr{D})$ stands for its average.

$$\begin{aligned}
[\![x_i := \mathcal{E}]\!]^{\mathscr{I}} &\stackrel{\text{def}}{=} \Big(\{x_i' = \mathcal{E}(x)\} \cup \{x_j' = x_j \mid j \ne i\}, \{\mathbb{E}[x_i'] = \mathcal{E}(x)\} \cup \{\mathbb{E}[x_j'] = x_j \mid j \ne i\}\Big) \\
[\![x_i \sim \mathscr{D}]\!]^{\mathscr{I}} &\stackrel{\text{def}}{=} \Big(\{\min(\mathscr{D}) \le x_i' \le \max(\mathscr{D})\} \cup \{x_j' = x_j \mid j \ne i\}, \{\mathbb{E}[x_i'] = \text{mean}(\mathscr{D})\} \cup \{\mathbb{E}[x_j'] = x_j \mid j \ne i\}\Big) \\
[\![\mathbf{skip}]\!]^{\mathscr{I}} &\stackrel{\text{def}}{=} \underline{1}_{\mathcal{I}}
\end{aligned}$$

Note we assume all expressions in the program are linear. For nonlinear arithmetic programs, one can adopt some linearization techniques [Farzan and Kincaid 2015; Miné 2006].

Theorem 5.4. *$\gamma_{\mathcal{I}}$ is a prob. over-abstraction from $\mathcal{C}$ to $\mathcal{I}$.*

Widening. Let $\triangledown$ be the standard widening operator on ordinary polyhedra [Halbwachs 1979]. Recall from Obs. 4.8 that whenever a widening operation $a \triangledown b$ is performed, the property $a \sqsubseteq_{\mathcal{A}} b$ holds. There is a subtle issue with expectation invariants when dealing with conditional or nondeterministic loops.

Observation 5.5. In a conventional program, if you have a loop "while B do S od," and I is a loop-invariant, then $I \wedge \neg B$ (which implies I) holds on exiting the loop. In contrast, for a conditional or nondeterministic loop in a probabilistic program, a loop-invariant that holds at the beginning and end of the loop body does not necessarily hold on exiting the loop.

Example 5.6. Consider the following program:

```
while $\neg(x = y)$ do
  if prob($\frac{1}{2}$) then $x := x + 1$ else $y := y + 1$ fi
od
```

For the loop body, we can derive an expectation invariant $\mathbb{E}[x' - y'] = x - y$; however, for the entire loop this property does not hold: at the end of the loop $x = y$ must hold, and hence $\mathbb{E}[x' - y']$ should be equal to 0.

Because of this issue, we use a pessimistic widening operator for conditional-choice and nondeterministic-choice: the widening operator forgets the expectation invariants and rebuilds them from standard invariants.

$$\begin{aligned}
(P_1, EP_1) \triangledown_c (P_2, EP_2) &\stackrel{\text{def}}{=} (P_1 \triangledown P_2, \mathbf{0} \sqcup P_2[\mathbb{E}[x']/x']) \\
(P_1, EP_1) \triangledown_n (P_2, EP_2) &\stackrel{\text{def}}{=} (P_1 \triangledown P_2, \mathbf{0} \sqcup P_2[\mathbb{E}[x']/x'])
\end{aligned}$$

We do not have a good method for $(P_1, EP_1) \triangledown_p (P_2, EP_2)$. We found that the following approach loses precision:

$$\textbf{let } P = (P_1 \triangledown P_2) \textbf{ in } (P, (EP_1 \triangledown EP_2) \sqcap (\mathbf{0} \sqcup P[\mathbb{E}[x']/x']))$$

In our experiments, we use $(P_1, EP_1) \triangledown_p (P_2, EP_2) \stackrel{\text{def}}{=} (P_1 \triangledown P_2, EP_2)$, which does no extrapolation in the $\mathcal{EP}$ component.

6 Evaluation

In this Section, we first describe the implementation of PMAF, and the three instantiations introduced in §5. Then, we evaluate the effectiveness and performance of the three analyses.

6.1 Implementation

PMAF is implemented in OCaml; the core framework consists of about 400 lines of code. The framework is implemented as a functor parametrized by a module representing a PMA, with some extra functions, such as widening and printing. This organization allows any analysis that can be formulated in PMAF to be implemented as a plugin. Also, the core framework relies on control-flow hyper-graphs, and provides users the flexibility to employ it with any front end. We use OCamlGraph [Conchon et al. 2007] as the implementation of fixed-point computation and Bourdoncle's algorithm.

The plugin for Bayesian inference is about 400 lines of code, including a lexer and a parser for the imperative language that we use in the examples of this paper. We use Lacaml [URL 2017a] to manipulate matrices. The plugins for the Markov decision problem with rewards and linear expectation-invariant analysis are about 200 lines and 500 lines, respectively. We use APRON [Jeannet and Miné 2009] for polyhedron operations. Most of the code in the plugins is to implement the PMA structure of the analysis domain.

Because of the numerical reasoning required when analyzing probabilistic programs, we need to be concerned about finite numerical precision in our implementations of the instantiations (although they are sound on a theoretical machine operating on reals). In our implementation, we use the fact that ascending chains of floating numbers always converge in a finite number of steps. The user could use the technique proposed by Darulova et al. [Darulova and Kuncak 2014] to obtain a sound guarantee on numerical precision.

6.2 Experiments

Evaluation Platform. Our experiments were performed on a machine with an Intel Core i5 2.4 GHz processor and 8GB of RAM under Mac OS X 10.13.4.

Bayesian Inference and Markov Decision Problem with Rewards. We tested our framework on Bayesian inference and Markov decision problem with rewards on handcrafted examples. The results of the evaluation of the two analyses are described in Tab. 1. The tables contains the number of lines; whether the program is non-recursive, tail-recursive, or recursive; the number of procedure calls; and the time taken by the implementation (measured by running each program 5 times and computing the 20% trimmed mean).

Table 1: Top: Bayesian inference. Bottom: Markov decision problem with rewards. (Time is in seconds.)

Program	#loc	rec?	#call	time
compare	17	n	0	2.22
dice	12	n	0	0.02
eg1	10	n	0	0.02
eg1-tail	16	t	2	0.02
eg2	10	n	0	0.02
eg2-tail	16	t	2	0.01
recursive	14	r	1	0.01
binary10	184	n	90	0.03
loop	10	n	0	0.03
quicksort7	109	n	42	0.03
recursive	13	t	1	0.03
student	43	t	8	0.03

Our framework computed the same answer (modulo floating-point round-off errors) as PReMo [URL 2017b], a tool for probabilistic recursive models. We did not compare with probabilistic abstract interpretation [Cousot and Monerau 2012] because its semantic foundation is substantially different from that of our framework—as we mentioned in §1, the order for resolving probabilistic behavior and nondeterministic behavior is different.

The analysis time of Bayesian inference grows exponentially with respect to the number of program variables.[6] The time cost comes from the explicit matrix representation of domain elements. One

[6]One should not assume that exponential growth makes the analysis useless; after all, predicate-abstraction

Table 2: Linear expectation-invariant analysis

Program	Expectation invariants	#loc	rec?	#call	time
2d-walk	$\mathbb{E}[x'] = x$, $\mathbb{E}[y'] = y$, $\mathbb{E}[dist'] = dist$, $\mathbb{E}[count'] \le count + 1$, $\mathbb{E}[count'] \ge count$	47	n	0	0.24
aggregate-rv	$\mathbb{E}[2x' - i'] = 2x - i$, $\mathbb{E}[x'] \le x + \frac{1}{2}$, $\mathbb{E}[x'] \ge x$	11	n	0	0.06
biased-coin	$\mathbb{E}[x'] \le x + \frac{1}{2}$, $\mathbb{E}[x'] \ge x - \frac{1}{2}$	25	n	0	0.06
binom-update ($p=\frac{1}{4}$)	$\mathbb{E}[4x' - n'] = 4x - n$, $\mathbb{E}[x'] \le x + \frac{1}{4}$, $\mathbb{E}[x'] \ge x$	14	n	0	0.06
coupon5	$\mathbb{E}[count' - i'] = count - i$ (1st), $\mathbb{E}[4count' - 5i'] = 4count - 5i$ (2nd), $\mathbb{E}[3count' - 5i'] = 3count - 5i$ (3rd), $\mathbb{E}[2count' - 5i'] = 2count - 5i$ (4th), $\mathbb{E}[count' - 5i'] = count - 5i$ (5th)	58	n	0	0.07
dist	$\mathbb{E}[x'] = x$, $\mathbb{E}[y'] = y$, $\mathbb{E}[z'] = \frac{1}{2}x + \frac{1}{2}y$	5	n	0	0.05
eg	$\mathbb{E}[x' + y'] = x + y + 3$, $\mathbb{E}[z'] = \frac{1}{4}z + \frac{3}{4}$, $\mathbb{E}[x'] \le x + 3$, $\mathbb{E}[x'] \ge x$	8	n	0	0.89
eg-tail	$\mathbb{E}[z'] \ge \frac{1}{4}z$, $\mathbb{E}[x'] \ge x$, $\mathbb{E}[y'] \ge y$, $\mathbb{E}[x' + y'] \ge x + y + \frac{3}{4}$	11	t	1	0.13
hare-turtle	$\mathbb{E}[2h' - 5t'] = 2h - 5t$, $\mathbb{E}[h'] \le h + \frac{5}{2}$, $\mathbb{E}[h'] \ge h$	15	n	0	0.06
hawk-dove	$\mathbb{E}[p1b' - count'] = p1b - count$, $\mathbb{E}[p2b' - count'] = p2b - count$, $\mathbb{E}[p1b'] \le p1b + 1$, $\mathbb{E}[p1b'] \ge p1b$	29	n	0	0.08
mot-ex	$\mathbb{E}[2x' - y'] = 2x - y$, $\mathbb{E}[4x' - 3count'] = 4x - 3count$, $\mathbb{E}[x'] \le x + \frac{3}{4}$, $\mathbb{E}[x'] \ge x$	16	n	0	0.06
recursive	$\mathbb{E}[x'] = x + 9$	13	r	2	0.37
uniform-dist	$\mathbb{E}[n'] \le 2n$, $\mathbb{E}[n'] \ge n$, $\mathbb{E}[g'] \le 2g + \frac{1}{2}$, $\mathbb{E}[g'] \ge g$	14	n	0	0.06

could use Algebraic Decision Diagrams [Bahar et al. 1997] as a compact representation to improve the efficiency.

The analyzer for the Markov decision problem with rewards works quickly and obtains some interesting results. *quicksort7* is a model of a randomized quicksort algorithm on an array of size 7 (obtained from [URL 2017b]), and our analysis results are consistent with the wost-case expected number of comparisons being $\Theta(n \log n)$.[7] *binary10* is a model of randomized binary search algorithm on an array of size 10, and our analysis results are consistent with the worst-case expected number of comparisons being $\Theta(\log n)$.

Linear Expectation-Invariant Analysis. We performed a more thorough evaluation of linear expectation-invariant analysis. We collected several examples from the literature on probabilistic invariant generation [Chakarov and Sankaranarayanan 2014; Katoen et al. 2010], and handcrafted some new examples to demonstrate particular capabilities of our domain, e.g., analysis of recursive programs. For the examples obtained from the loop-invariant-generation benchmark, we extracted the loop body as our test programs. Also, we performed a positive-negative decomposition to make sure all program variables are nonnegative. That is, we represented each variable x as $x^+ - x^-$ where $x^+, x^- \ge 0$, and replaced every operation on variables with appropriate operations on the decomposed variables.

The results of the evaluation are shown in Tab. 2, which lists the expectation invariants obtained, and the time taken by the implementation. In general, the analysis runs quickly—all the examples are processed in less than one second. The analysis time mainly depends on the number of program variables and the size of the control-flow hyper-graph.

As shown in Tab. 2, our analysis can derive nontrivial expectation invariants, e.g., relations among different program variables such as $\mathbb{E}[x' + y'] = x + y + 3$, $\mathbb{E}[2x' - y'] = 2x - y$. In most cases, our results are at least as precise as those in [Chakarov and Sankaranarayanan 2014; Katoen et al. 2010]. Exceptions are *biased-coin* and *uniform-dist*, collected from [Katoen et al. 2010], where their invariant-generation algorithm uses a template-based approach and the form of expectations can be more complicated, e.g., $[P_1] \cdot \mathcal{E}_1 + [P_2] \cdot \mathcal{E}_2$ where P_1, P_2 are linear assertions and $\mathcal{E}_1, \mathcal{E}_2$ are linear expressions. Nevertheless, our analysis is fully automated and applicable to general programs, while [Katoen et al.

domains [Graf and Saïdi 1997] also grow exponentially: the universe of assignments to a set of Boolean variables grows exponentially in the number of variables. Finding useful coarser abstractions for Bayesian inference—by analogy with the techniques of Ball et al. [Ball et al. 2001] for predicate abstraction—might be an interesting direction for future work.

[7]The analysis computes *worst-case expected number* because the underlying semantics resolves nondeterminism first and probabilistic-choice second, and thus the analysis computes $\max_{\text{nondet. resolution}} \mathbb{E}[\text{\#comparisons under resolution}]$.

2010] requires interactive proofs for nested loops, and [Chakarov and Sankaranarayanan 2014] works only for single loops.

7 Related Work

Static Analysis for Standard Programs. Our framework is an extension of interprocedural dataflow analysis [Knoop and Steffen 1992; Lal et al. 2005; Müller-Olm and Seidl 2004; Sharir and Pnueli 1981] to probabilistic programs, but it does not support some language features that standard dataflow analysis has been used to address, e.g., calls through function pointers.

Compared to the Galois connections that are ordinarily used in abstract interpretation [Cousot and Cousot 1977, 1979], our definition of probabilistic abstractions is based on just a concretization function, so PMAF does not have the full power of standard abstract-interpretation machinery.

Static Analysis for Probabilistic Programs. Most closely related to our work is probabilistic abstract interpretation [Cousot and Monerau 2012; Monniaux 2000, 2001, 2003], which is discussed in the introduction. There is a long line of research on manual reasoning techniques for probabilistic programs [Ferrer Fioriti and Hermanns 2015; Kaminski et al. 2016; Kozen 1985; McIver and Morgan 2001; Olmedo et al. 2016]. The main difference to this work is that we focus on the design and implementation of automatic techniques that that rely on computing fixed points.

Other work focuses on specialized automatic analyses for specific properties. Claret et al. [Claret et al. 2013] proposed a dataflow analysis for Bayesian inference on Boolean programs that we reformulate in PMAF to lift it to the interprocedural level. There are different techniques for automatically proving termination, such as probabilistic pushdown automata [Brázdil et al. 2014, 2015] and martingales and stochastic invariants [Chatterjee et al. 2016b, 2017]. Martingales for automatic analysis of probabilistic programs have been pioneered by Chakarov et al. [Chakarov and Sankaranarayanan 2013]. Compared with existing techniques for probabilistic invariant generation [Barthe et al. 2016a; Chakarov and Sankaranarayanan 2013, 2014; Chatterjee et al. 2017], the expectation-invariant analysis proposed in §5.3 is designed as a two-vocabulary domain utilizing the well-studied polyhedral abstract domain.

Semantics for Probabilistic Programs. There is a long tradition of using probability kernels to define the semantics of probabilistic programs. Kernels were used by Kozen [Kozen 1985] to give a semantics for Probabilistic Propositional Dynamic Logic (PPDL), a probabilistic generalization of PDL. Kozen considers well-structured programs with sequencing and conditional-choice, but without non-deterministic choice. He does not consider reasoning methods that use abstract interpretation of his PPDL semantics. There are a list of domain-theoretic studies on probabilistic nondeterminism [den Hartog and de Vink 1999; Jones 1989; Jones and Plotkin 1989; Mislove 2000; Mislove et al. 2004; Tix et al. 2009], which develop powerdomain constructions over probability distributions, but do not consider powerdomains over kernels. Borgström et al. [Borgström et al. 2016] have used kernels to define the operational semantics of a probabilistic lambda calculus. The main novelty of our denotational semantics in §3.3 is that it is defined for control-flow hyper-graphs, based on kernels.

Other Analyses Based on Hyper-Graphs. Hyper-graph-based analyses go back to the join-over-all-hyper-path-valuations of Knuth [Knuth 1977]. Other analyses based on hyper-graphs includes Möncke and Wilhelm's [Möncke and Wilhelm 1991] framework for finding join-over-all-hyper-path-valuations for *partially ordered* abstract domains. In the hyper-paths in this paper, we use binary hyper-edges to model calls, as well as conditional, probabilistic, and nondeterministic choice. For *acyclic* hyper-graphs, Eisner has considered semirings for computing expectations and variances of random variables [Li and Eisner 2009]. He works with a discrete sample space: all hyper-paths in a given hyper-graph, and the value of a random variable for a given hyper-path is built up as the sum of the values contributed by each hyper-edge. In our work, we consider *cyclic* hyper-graphs, and the nature of the computation that a hyper-path represents is more complex than that considered by

Eisner.

Acknowledgments

This article is based on research supported, in part, by a gift from Rajiv and Ritu Batra; by AFRL under DARPA MUSE award FA8750-14-2-0270, DARPA STAC award FA8750-15-C-0082, and DARPA award FA8750-16-2-0274; and by the UW-Madison Office of the Vice Chancellor for Research and Graduate Education with funding from the Wisconsin Alumni Research Foundation. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the sponsoring agencies.

We thank the nurses from the Univ. of Pittsburgh Medical Center for their professional medical care, which was instrumental in finishing the submission before the PLDI deadline.

References

2017a. Lacaml - Linear Algebra for OCaml. https://github.com/mmottl/lacaml.

2017b. PReMo – Probabilistic Recursive Models analyzer. http://groups.inf.ed.ac.uk/premo/.

Samson Abramsky and Achim Jung. 1994. Domain Theory. In *Handbook of Logic in Computer Science*. Vol. 3. Oxford University Press Oxford, UK.

R. Iris Bahar, Erica A. Frohm, Charles M. Gaona, Gary D. Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. 1997. Algebraic Decision Diagrams and their Applications. *Formal Methods in System Design* 10 (1997), 171–206.

T. Ball, A. Podelski, and S.K. Rajamani. 2001. Boolean and Cartesian Abstraction for Model Checking C Programs. In *Tools and Algs. for the Construct. and Anal. of Syst.*

Gilles Barthe, Thomas Espitau, Luis María Ferrer Fioriti, and Justin Hsu. 2016a. Synthesizing Probabilistic Invariants via Doob's Decomposition. In *Computer Aided Verification*. 43–61.

Gilles Barthe, Thomas Espitau, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2016b. *Formal Certification of Randomized Algorithms*. Technical Report. http://justinh.su/files/papers/ellora.pdf.

Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. 2009. Formal Certification of Code-based Cryptographic Proofs. In *Proceedings of the 36th Annual Symposium on Principles of Programming Languages (POPL'09)* (Savannah, GA, USA). ACM, New York, NY, USA, 90–101.

Patrick Billingsley. 2012. *Probability and Measure*. John Wiley & Sons, Inc.

Johannes Borgström, Ugo Dal Lago, Andrew D. Gordon, and Marcin Szymczak. 2016. A Lambda-Calculus Foundation for Universal Probabilistic Programming. In *International Conference on Functional Programming*. 33–46.

Francois Bourdoncle. 1993. Efficient Chaotic Iteration Strategies With Widenings. In *Formal Methods in Prog. and Their Applications*.

Tomáš Brázdil, Stefan Kiefer, and Antonín Kučera. 2014. Efficient Analysis of Probabilistic Programs with an Unbounded Counter. *J. ACM* 61, 41 (2014).

Tomáš Brázdil, Stefan Kiefer, Antonín Kučera, and Ivana Hutařová Vařeková. 2015. Runtime Analysis of Probabilistic Programs with Unbounded Recursion. *J. Comput. System Sci.* 81 (2015), 288–310.

Bob Carpenter, Andrew Gelman, Matthew D. Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2017. Stan: A Probabilistic Programming Language. *Journal of Statistical Software* 76 (2017).

Aleksandar Chakarov and Sriram Sankaranarayanan. 2013. Probabilistic Program Analysis with Martingales. In *Computer Aided Verification*. 511–526.

Aleksandar Chakarov and Sriram Sankaranarayanan. 2014. Expectation Invariants for Probabilistic Program Loops as Fixed Points. In *Static Analysis Symposium*. 85–100.

Krishnendu Chatterjee, Hongfei Fu, and Amir Kafshdar Goharshady. 2016a. Termination Analysis of Probabilistic Programs Through Positivstellensatz's. In *Computer Aided Verification - 28th International Conference (CAV'16)*. 3–22.

Krishnendu Chatterjee, Hongfei Fu, Petr Novotný, and Rouzbeh Hasheminezhad. 2016b. Algorithmic Analysis of Qualitative and Quantitative Termination Problems for Affine Probabilistic Programs. In *Principles of Programming Languages*. 327–342.

Krishnendu Chatterjee, Petr Novotný, and Đorđe Žikelić. 2017. Stochastic Invariants for Probabilistic Termination. In *Principles of Programming Languages*. 145–160.

Guillaume Claret, Sriram K. Rajamani, Aditya V. Nori, Andrew D. Gordon, and Johannes Borgström. 2013. Bayesian Inference using Data Flow Analysis. In *Foundations of Software Engineering*. 92–102.

Sylvain Conchon, Jean-Christophe Filliâtre, and Julien Signoles. 2007. Designing a Generic Graph Library Using ML Functors. In *Trends in Functional Programming*.

P. Cousot. 1981. Semantic Foundations of Program Analysis. In *Program Flow Analysis: Theory and Applications*, S.S. Muchnick and N.D. Jones (Eds.). Prentice-Hall, Englewood Cliffs, NJ, Chapter 10, 303–342.

P. Cousot and R. Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Princ. of Prog. Lang.* 238–252.

P. Cousot and R. Cousot. 1978. Static Determination of Dynamic Properties of Recursive Procedures. In *Formal Descriptions of Programming Concepts, (IFIP WG 2.2, St. Andrews, Canada, August 1977)*, E.J. Neuhold (Ed.). North-Holland, 237–277.

P. Cousot and R. Cousot. 1979. Systematic Design of Program Analysis Frameworks. In *Princ. of Prog. Lang.* 269–282.

P. Cousot and N. Halbwachs. 1978. Automatic Discovery of Linear Constraints Among Variables of a Program. In *Princ. of Prog. Lang.* 84–96.

Patrick Cousot and Michael Monerau. 2012. Probabilistic Abstract Interpretation. In *European Symposium on Programming*. 166–190.

Eva Darulova and Viktor Kuncak. 2014. Sound Compilation of Reals. In *Principles of Programming Languages*.

J.I. den Hartog and E.P. de Vink. 1999. Mixing Up Nondeterminism and Probability: a preliminary report. *Electronic Notes in Theoretical Computer Science* 22 (1999), 88–110.

E.W. Dijkstra. 1976. *A Discipline of Programming*. Prentice-Hall.

Kousha Etessami, Dominik Wojtczak, and Mihalis Yannakakis. 2008. Recursive Stochastic Games with Positive Rewards. In *International Colloquium on Automata, Languages, and Programming*. 711–723.

Kousha Etessami and Mihalis Yannakakis. 2005. Recursive Markov Chains, Stochastic Grammars, and Monotone Systems of Nonlinear Equations. In *Symposium on Theoretical Aspects of Computer Science*. 340–352.

Kousha Etessami and Mihalis Yannakakis. 2015. Recursive Markov Decision Processes and Recursive Stochastic Games. *J. ACM* 62, 11 (2015).

Azadeh Farzan and Zachary Kincaid. 2015. Compositional Recurrence Analysis. In *Formal Methods in Compiter-Aided Design*.

Luis María Ferrer Fioriti and Holger Hermanns. 2015. Probabilistic Termination: Soundness, Completeness, and Compositionality. In *Principles of Programming Languages*. 489–501.

R.W. Floyd. 1967. Assigning Meanings to Programs. In *Proc. AMS Symposium in Appl. Math.*, Vol. 19.

G. Gallo, G. Longo, S. Pallottino, and S. Nguyen. 1993. Directed Hypergraphs and Applications. *Disc. Appl. Math.* 42 (1993), 177–201.

Timon Gehr, Sasa Misailovic, and Martin T. Vechev. 2016. PSI: Exact Symbolic Inference for Probabilistic Programs. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*. 62–83.

Zoubin Ghahramani. 2015. Probabilistic machine learning and artificial intelligence. *Nature* 521 (2015), 452–459.

Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, and Joshua B. Tenenbaum. 2008. Church: a language for generative models, In Uncertainty in Artificial Intelligence. *Special Issue: Conference on Practical Bayesian Statistics*, 220–229.

Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. 2014. Probabilistic Programming. In *Proceedings of the on Future of Software Engineering (FOSE'14)*. 167–181.

S. Graf and H. Saïdi. 1997. Construction of Abstract State Graphs with PVS. In *Computer Aided Verif.*

Carl A. Gunter, Peter D. Mosses, and Dana S. Scott. 1989. *Semantic Domains and Denotational Semantics*. Technical Report. University of Pennsylvania Department of Computer and Information Science.

N. Halbwachs. 1979. Détermination automatique de relations linéaires vérifiées par les variables d'un programme. Thèse de 3e cycle. (As cited in [Halbwachs et al. 1997].).

N. Halbwachs, Y.-E. Proy, and P. Roumanoff. 1997. Verification of Real-Time Systems Using Linear Relation Analysis. *Formal Methods in System Design* 11, 2 (1997), 157–185.

C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12 (1969), 576–580.

S. Horwitz, T. Reps, and M. Sagiv. 1995. Demand Interprocedural Dataflow Analysis. In *Found. of Softw. Eng.* 104–115.

B. Jeannet and A. Miné. 2009. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *Computer Aided Verif.*

Claire Jones. 1989. *Probabilistic Non-determinism*. Ph.D. Dissertation. University of Edinburgh Edinburgh, Scotland, UK.

Claire Jones and Gordon Plotkin. 1989. A Probabilistic Powerdomain of Evaluations. In *Logic in Computer Science*. 186–195.

Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. 2016. Weakest Precondition Reasoning for Expected Run—Times of Probabilistic Programs. In *European Symposium on Programming*. 364–389.

Joost Pieter Katoen, Annabelle K. McIver, Larissa A. Meinicke, and Carroll C. Morgan. 2010. Linear-Invariant Generation for Probabilistic Programs: Automated Support for Proof-Based Methods. In *Static Analysis Symposium*. 390–406.

Mark Kattenbelt, Marta Kwiatkowska, Gethin Norman, and David Parker. 2009. Abstraction Refinement for Probabilistic Software. In *Verification, Model Checking, and Abstract Interpretation*. 182–197.

G.A. Kildall. 1973. A Unified Approach to Global Program Optimization. In *Princ. of Prog. Lang.* 194–206.

J. Knoop and B. Steffen. 1992. The Interprocedural Coincidence Theorem. In *Comp. Construct.* 125–140.

D.E. Knuth. 1977. A Generalization of Dijkstra's Algorithm. *Inf. Proc. Let.* 6, 1 (1977), 1–5.

Stanley Kok, Marc Sumner, Matthew Richardson, Parag Singla, Hoifung Poon, Daniel Lowd, Jue Wang, and Pedro Domingos. 2007. *The Alchemy System for Statistical Relational AI*. Technical Report. University of Washington.

Dexter Kozen. 1981. Semantics of Probabilistic Programs. *J. Comput. System Sci.* 22 (1981), 328–350.

Dexter Kozen. 1985. A Probabilistic PDL. *J. Comput. System Sci.* 30 (1985), 162–178.

A. Lal, T. Reps, and G. Balakrishnan. 2005. Extended Weighted Pushdown Systems. In *Computer Aided Verif.*

Z. Li and J. Eisner. 2009. First- and Second-Order Expectation Semirings with Applications to Minimum-Risk Training on Translation Forests. In *EMNLP*.

Annabelle K. McIver and Carroll C. Morgan. 2001. Partial correctness for probabilistic demonic programs. *Theoretical Computer Science* 266 (2001), 513–541.

Annabelle K. McIver and Carroll C. Morgan. 2005. *Abstraction, Refinement and Proof for Probabilistic Systems*. Springer Science+Business Media, Inc.

Brian Milch, Bhaskara Marthi, Stuart Russell, David Sontag, Daniel L. Ong, and Andrey Kolobov. 2005. BLOG: Probabilistic Models with Unknown Objects. In *International Joint Conference on Artificial Intelligence*. 1352–1359.

Antoine Miné. 2006. Symbolic Methods to Enhance the Precision of Numerical Abstract Domains. In *Verif., Model Checking, and Abs. Interp.*

T. Minka, J.M. Winn, J.P. Guiver, S. Webster, Y. Zaykov, B. Yangel, A. Spengler, and J. Bronskill. 2014. Infer.NET 2.6. Microsoft Research Cambridge. http://research.microsoft.com/infernet.

Michael Mislove. 2000. Nondeterminism and Probabilistic Choice: Obeying the Laws. In *Concurrency Theory*. 350–365.

Michael Mislove, Joël Ouaknine, and James Worrell. 2004. Axioms for Probability and Nondeterminism. *Electronic Notes in Theoretical Computer Science* 96 (2004), 7–28.

U. Möncke and R. Wilhelm. 1991. Grammar Flow Analysis. In *Attribute Grammars, Applications and Systems, (Int. Summer School SAGA) (Lec. Notes in Comp. Sci.)*, H. Alblas and B. Melichar (Eds.), Vol. 545. Springer-Verlag, Prague, Czechoslovakia, 151–186.

David Monniaux. 2000. Abstract Interpretation of Probabilistic Semantics. In *Static Analysis Symposium*. 322–339.

David Monniaux. 2001. Backwards Abstract Interpretation of Probabilistic Programs. In *European Symposium on Programming*. 367–382.

David Monniaux. 2003. Abstract Interpretation of Programs as Markov Decision Processes. In *Static Analysis Symposium*. 237–254.

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

Federico Olmedo, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. 2016. Reasoning about Recursive Probabilistic Programs. In *Logic in Computer Science*. 672–681.

Prakash Panangaden. 1999. The Category of Markov Kernels. *Electronic Notes in Theoretical Computer Science* 22, Supplement C (1999).

Avi Pfeffer. 2005. *The Design and Implementation of IBAL: A General-Purpose Probabilistic Language*. Technical Report. Harvard Computer Science Group.

Martin L. Puterman. 1994. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc.

G. Ramalingam. 1996. *Bounded Incremental Computation*. Springer-Verlag.

T. Reps, S. Horwitz, and M. Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Princ. of Prog. Lang.* 49–61.

M. Sagiv, T. Reps, and S. Horwitz. 1996. Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation. *Theor. Comp. Sci.* 167 (1996), 131–170.

Sriram Sankaranarayanan, Aleksandar Chakarov, and Sumit Gulwani. 2013. Static Analysis for Probabilistic Programs: Inferring Whole Program Properties from Finitely Many Paths. In *Programming Language Design and Implementation*. 447–458.

M. Sharir and A. Pnueli. 1981. Two Approaches to Interprocedural Data Flow Analysis. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall.

R.E. Tarjan. 1981. A Unified Approach to Path Problems. *J. ACM* 28, 3 (1981), 577–593.

Regina Tix, Klaus Keimel, and Gordon Plotkin. 2009. Semantic Domains for Combining Probability and Non-Determinism. *Electronic Notes in Theoretical Computer Science* 222 (2009), 3–99.

D. Wang, J. Hoffman, and T. Reps. 2018. A Denotational Semantics for Nondeterminism in Probabilistic Programs. Available at www.cs.cmu.edu/~diw3/papers/WangHR18.pdf.

A Proofs

Readers can refer to Abramsky and Jung [1994]; Tix et al. [2009]; Wang et al. [2018] for the domain-theoretic notions we will use in the proofs.

A.1 Thm. 4.7

Proof. We only show the proof for the over-approximations. By definition, $\mathscr{C}[P] = \mathrm{lfp}^{\dot{\sqsubseteq}_{\mathcal{C}}}_{\lambda v.\underline{\perp}_{\mathcal{C}}} F^{\mathscr{C}}_P = \sup\{(F^{\mathscr{C}}_P)^n(\lambda v.\underline{\perp}_{\mathcal{C}})\}$ by Kleene, and $\mathscr{A}[P] = \mathrm{lfp}^{\dot{\sqsubseteq}_{\mathcal{A}}}_{\lambda v.\underline{\perp}_{\mathcal{A}}} F^{\mathscr{A}}_P$ obtained by Knaster-Tarski. We want to show that for all n we have $(F^{\mathscr{C}}_P)^n(\lambda v.\underline{\perp}_{\mathcal{C}}) \dot{\sqsubseteq}_{\mathcal{C}} \dot{\gamma}(\mathscr{A}[P])$. Let's prove by induction on n. The base case follows directly from the fact that $\underline{\perp}_{\mathcal{C}}$ is the least element in $\mathcal{C}$. Suppose we know $(F^{\mathscr{C}}_P)^n(\lambda v.\underline{\perp}_{\mathcal{C}}) \dot{\sqsubseteq}_{\mathcal{C}} \dot{\gamma}(\mathscr{A}[P])$ for some n. Let's denote the left hand side by LHS and $\mathscr{A}[P]$ by SOL. We want to show that $F^{\mathscr{C}}_P(LHS) \dot{\sqsubseteq}_{\mathcal{C}} \dot{\gamma}(SOL)$. This expands to $F^{\mathscr{C}}_P(LHS)(v) \sqsubseteq_{\mathcal{C}} \gamma(SOL(v))$ for all $v \in V$. We proceed by a case analysis on the kind of edges leaving v.

1. If $v = v_i^{\mathsf{exit}}$ for some i, then $F^{\mathscr{C}}_P(LHS)(v) = \underline{1}_{\mathcal{C}}$. Then we can conclude this case by showing that $SOL(v) = \underline{1}_{\mathcal{A}}$. By definition of SOL, we know that $F^{\mathscr{A}}_P(SOL) = SOL$, thus $F^{\mathscr{A}}_P(SOL)(v) = SOL(v)$. By definition of $F^{\mathscr{A}}_P$, we know that $F^{\mathscr{A}}_P(SOL)(v) = \underline{1}_{\mathcal{A}}$.
2. If $v \neq v_i^{\mathsf{exit}}$ for all i, then v is associated with some $e = \langle v, \{u_1, \cdots, u_k\}\rangle \in E$, and we have

$$\begin{aligned} F^{\mathscr{C}}_P(LHS)(v) &= \widehat{Ctrl(e)}(LHS(u_1), \cdots, LHS(u_k)) \\ &\sqsubseteq_{\mathcal{C}} \widehat{Ctrl(e)}(\gamma(SOL(u_1)), \cdots, \gamma(SOL(u_k))). \end{aligned}$$

   If we can prove that for any kind of $Ctrl(e)$ it holds that $\widehat{Ctrl(e)}(\gamma(x_1), \cdots, \gamma(x_k)) \sqsubseteq_{\mathcal{C}} \gamma(\widehat{Ctrl(e)}^{\sharp}(x_1, \cdots, x_k))$, then we can conclude the case by the following argument:

$$\begin{aligned} F^{\mathscr{C}}_P(LHS)(v) &\sqsubseteq_{\mathcal{C}} \gamma(\widehat{Ctrl(e)}^{\sharp}(SOL(u_1), \cdots, SOL(u_k))) \\ &= \gamma(F^{\mathscr{A}}_P(SOL)(v)) \\ &= \gamma(SOL(v)). \end{aligned}$$

   Now consider the form of $Ctrl(e)$.

   - $Ctrl(e) = seq[\mathrm{act}]$: We want to show that $\widehat{seq[\mathrm{act}]}(\gamma(x_1)) \sqsubseteq_{\mathcal{C}} \gamma(\widehat{seq[\mathrm{act}]}^{\sharp}(x_1))$. It is equivalent to $[\![\mathrm{act}]\!]^{\mathscr{C}} \otimes_{\mathcal{C}} \gamma(x_1) \sqsubseteq_{\mathcal{C}} \gamma([\![\mathrm{act}]\!]^{\mathscr{A}} \otimes_{\mathcal{A}} x_1)$. Indeed, we have

$$[\![\mathrm{act}]\!]^{\mathscr{C}} \otimes_{\mathcal{C}} \gamma(x_1) \sqsubseteq_{\mathcal{C}} \gamma([\![\mathrm{act}]\!]^{\mathscr{A}}) \otimes_{\mathcal{C}} \gamma(x_1) \sqsubseteq_{\mathcal{C}} \gamma([\![\mathrm{act}]\!]^{\mathscr{A}} \otimes_{\mathcal{A}} x_1)$$

     by assumption, monotonicity of $\otimes_{\mathcal{C}}$, and properties of γ.

   - $Ctrl(e) = call[i \to j]$: We want to show that $\widehat{call[i \to j]}(\gamma(x_1)) \sqsubseteq_{\mathcal{C}} \gamma(\widehat{call[i \to j]}^{\sharp}(x_1))$. It is equivalent to $LHS(v_j^{\mathsf{entry}}) \otimes_{\mathcal{C}} \gamma(x_1) \sqsubseteq_{\mathcal{C}} \gamma(SOL(v_j^{\mathsf{entry}}) \otimes_{\mathcal{A}} x_1)$. Indeed, we have

$$LHS(v_j^{\mathsf{entry}}) \otimes_{\mathcal{C}} \gamma(x_1) \sqsubseteq_{\mathcal{C}} \gamma(SOL(v_j^{\mathsf{entry}})) \otimes_{\mathcal{C}} \gamma(x_1) \sqsubseteq_{\mathcal{C}} \gamma(SOL(v_j^{\mathsf{entry}}) \otimes_{\mathcal{A}} x_1)$$

     by induction hypothesis, monotonicity of $\otimes_{\mathcal{C}}$, and properties of γ.

   - $Ctrl(e) = cond[\varphi]$: We want to show that $\widehat{cond[\varphi]}(\gamma(x_1), \gamma(x_2)) \sqsubseteq_{\mathcal{C}} \gamma(\widehat{cond[\varphi]}^{\sharp}(x_1, x_2))$. It is equivalent to $\gamma(x_1)\,{}_{\varphi}\diamond_{\mathcal{C}}\,\gamma(x_2) \sqsubseteq_{\mathcal{C}} \gamma(x_1\,{}_{\varphi}\diamond_{\mathcal{A}}\,x_2)$. Appeal to properties of γ.

- $Ctrl(e) = prob[p]$: We want to show that $\widehat{prob[p]}(\gamma(x_1),\gamma(x_2)) \sqsubseteq_{\mathcal{C}} \gamma(\widehat{prob[p]}^{\sharp}(x_1,x_2))$. It is equivalent to $\gamma(x_1)\,{}_p{\oplus}_{\mathcal{C}}\,\gamma(x_2) \sqsubseteq_{\mathcal{C}} \gamma(x_1\,{}_p{\oplus}_{\mathcal{A}}\,x_2)$. Appeal to properties of γ.
- $Ctrl(e) = ndet$: We want to show that $\widehat{ndet}(\gamma(x_1),\gamma(x_2)) \sqsubseteq_{\mathcal{C}} \gamma(\widehat{ndet}^{\sharp}(x_1,x_2))$. It is equivalent to $\gamma(x_1) \Cup_{\mathcal{C}} \gamma(x_2) \sqsubseteq_{\mathcal{C}} \gamma(x_1 \Cup_{\mathcal{A}} x_2)$. Appeal to properties of γ.

□

A.2 Thm. 5.1

Proof. By the properties of the concrete semantics, we have $\gamma_{\mathcal{B}}(a) = \uparrow(\lambda(s,S').\sum_{s'\in S'} a(s,s'))$, i.e., the upper closure of $\{a\}$ in $\mathbb{P}\Omega$.

- We want to show $\gamma_{\mathcal{B}}(\underline{1}_{\mathcal{B}}) \sqsubseteq_{\mathcal{C}} \underline{1}_{\mathcal{C}}$. Appeal to the fact that $\uparrow(\lambda(s,S').[s\in S']) = \{\underline{1}_{\mathbb{K}}\} = \underline{1}_{\mathbb{P}}$.
- We want to show for all $Q_1,Q_2 \in \mathcal{B}$, $\gamma_{\mathcal{B}}(Q_1 \otimes_{\mathcal{B}} Q_2) \sqsubseteq_{\mathcal{C}} \gamma_{\mathcal{B}}(Q_1) \otimes_{\mathcal{C}} \gamma_{\mathcal{B}}(Q_2)$. It is sufficient to show $\gamma_{\mathcal{B}}(Q_1 \times Q_2) \supseteq \uparrow conv(\gamma_{\mathcal{B}}(Q_1) \otimes \gamma_{\mathcal{B}}(Q_2))$. Observe that $\gamma_{\mathcal{B}}(Q_1 \times Q_2)$ is saturated and convex, it is sufficient to show $\gamma_{\mathcal{B}}(Q_1) \otimes \gamma_{\mathcal{B}}(Q_2) \subseteq \gamma_{\mathcal{B}}(Q_1 \times Q_2)$. Suppose $\kappa_1 \in \gamma_{\mathcal{B}}(Q_1)$ and $\kappa_2 \in \gamma_{\mathcal{B}}(Q_2)$. Let $q_1 = \lambda(s,S').\sum_{s'\in S'} Q_1(s,s')$ and $q_2 = \lambda(s,S').\sum_{s'\in S'} Q_2(s,s')$. Then $\kappa_1 \sqsupseteq q_1$ and $\kappa_2 \sqsupseteq q_2$. Because $\otimes$ is monotone, we know that $\kappa_1 \otimes \kappa_2 \sqsupseteq q_1 \otimes q_2$. Observe that $q_1 \otimes q_2 = \lambda(s,S').\int q_1(s,dy)q_2(y,S') = \lambda(s,S').\sum_{s'\in S'}(Q_1\times Q_2)(s,s')$, thus $q_1 \otimes q_2 \in \gamma_{\mathcal{B}}(Q_1 \times Q_2)$. Hence $\kappa_1 \otimes \kappa_2 \in \gamma_{\mathcal{B}}(Q_1 \times Q_2)$.
- We want to show for all $Q_1,Q_2 \in \mathcal{B}$ and $\varphi \in \mathcal{L}$, $\gamma_{\mathcal{B}}(Q_1\,{}_{\varphi}{\diamond}_{\mathcal{B}}\,Q_2) \sqsubseteq_{\mathcal{C}} \gamma_{\mathcal{B}}(Q_1)\,{}_{\varphi}{\diamond}_{\mathcal{C}}\,\gamma_{\mathcal{B}}(Q_2)$. It is sufficient to show $\gamma_{\mathcal{B}}(\lambda(s,s').\textbf{if } \widehat{\varphi}(s) \textbf{ then } Q_1(s,s') \textbf{ else } Q_2(s,s')) \supseteq \uparrow \gamma_{\mathcal{B}}(Q_1)\,{}_{\varphi}{\diamond}\,\gamma_{\mathcal{B}}(Q_2)$. Observe that the left-hand-side is saturated, it is sufficient to show $\gamma_{\mathcal{B}}(Q_1)\,{}_{\varphi}{\diamond}\,\gamma_{\mathcal{B}}(Q_2) \subseteq \gamma_{\mathcal{B}}(\lambda(s,s').\textbf{if } \widehat{\varphi}(s) \textbf{ then } Q_1(s,s') \textbf{ else } Q_2(s,s'))$. Suppose $\kappa_1 \in \gamma_{\mathcal{B}}(Q_1)$ and $\kappa_2 \in \gamma_{\mathcal{B}}(Q_2)$. Let $q_1 = \lambda(s,S').\sum_{s'\in S'} Q_1(s,s')$ and $q_2 = \lambda(s,S').\sum_{s'\in S'} Q_2(s,s')$. Then $\kappa_1 \sqsupseteq q_1$ and $\kappa_2 \sqsupseteq q_2$. Because ${}_{\varphi}{\diamond}$ is monotone, we know that $\kappa_1\,{}_{\varphi}{\diamond}\,\kappa_2 \sqsupseteq q_1\,{}_{\varphi}{\diamond}\,q_2$. Observe that $q_1\,{}_{\varphi}{\diamond}\,q_2 = \lambda(s,S').\textbf{if } \widehat{\varphi}(s) \textbf{ then } q_1(s,S') \textbf{ else } q_2(s,S') = \lambda(s,S').\sum_{s'\in S'} \textbf{if } \widehat{\varphi}(s) \textbf{ then } Q_1(s,s') \textbf{ else } Q_2(s,s')$, thus $q_1\,{}_{\varphi}{\diamond}\,q_2 \in \gamma_{\mathcal{B}}(\lambda(s,s').\textbf{if } \widehat{\varphi}(s) \textbf{ then } Q_1(s,s') \textbf{ else } Q_2(s,s'))$. Hence $\kappa_1\,{}_{\varphi}{\diamond}\,\kappa_2 \in \gamma_{\mathcal{B}}(\lambda(s,s').\textbf{if } \widehat{\varphi}(s) \textbf{ then } Q_1(s,s') \textbf{ else } Q_2(s,s'))$.
- We want to show for all $Q_1,Q_2 \in \mathcal{B}$ and $p \in [0,1]$, $\gamma_{\mathcal{B}}(Q_1\,{}_p{\oplus}_{\mathcal{B}}\,Q_2) \sqsubseteq_{\mathcal{C}} \gamma_{\mathcal{B}}(Q_1)\,{}_p{\oplus}_{\mathcal{C}}\,\gamma_{\mathcal{B}}(Q_2)$. It is sufficient to show $\gamma_{\mathcal{B}}(pQ_1 + (1-p)Q_2) \supseteq \uparrow \gamma_{\mathcal{B}}(Q_1)\,{}_p{\oplus}\,\gamma_{\mathcal{B}}(Q_2)$. Observe that the left-hand-side is saturated, it is sufficient to show $\gamma_{\mathcal{B}}(Q_1)\,{}_p{\oplus}\,\gamma_{\mathcal{B}}(Q_2) \subseteq \gamma_{\mathcal{B}}(pQ_1 + (1-p)Q_2)$. Suppose $\kappa_1 \in \gamma_{\mathcal{B}}(Q_1)$ and $\kappa_2 \in \gamma_{\mathcal{B}}(Q_2)$. Let $q_1 = \lambda(s,S').\sum_{s'\in S'} Q_1(s,s')$ and $q_2 = \lambda(s,S').\sum_{s'\in S'} Q_2(s,s')$. Then $\kappa_1 \sqsupseteq q_1$ and $\kappa_2 \sqsupseteq q_2$. Because ${}_p{\oplus}$ is monotone, we know that $\kappa_1\,{}_p{\oplus}\,\kappa_2 \sqsupseteq q_1\,{}_p{\oplus}\,q_2$. Observe that $q_1\,{}_p{\oplus}\,q_2 = \lambda(s,S').p\cdot q_1(s,S') + (1-p)\cdot q_2(s,S') = \lambda(s,S').\sum_{s'\in S'} p\cdot Q_1(s,s') + (1-p)\cdot Q_2(s,s')$, thus $q_1\,{}_p{\oplus}\,q_2 \in \gamma_{\mathcal{B}}(pQ_1 + (1-p)Q_2)$. Hence $\kappa_1\,{}_p{\oplus}\,\kappa_2 \in \gamma_{\mathcal{B}}(pQ_1 + (1-p)Q_2)$.
- We want to show that for all $Q_1,Q_2 \in \mathcal{B}$, $\gamma_{\mathcal{B}}(Q_1 \Cup_{\mathcal{B}} Q_2) \sqsubseteq_{\mathcal{C}} \gamma_{\mathcal{B}}(Q_1) \Cup_{\mathcal{C}} \gamma_{\mathcal{B}}(Q_2)$. It is sufficient to show $\gamma_{\mathcal{B}}(\dot{\min}(Q_1,Q_2)) \supseteq \uparrow conv(\gamma_{\mathcal{B}}(Q_1) \cup \gamma_{\mathcal{B}}(Q_2))$. Observe that the left-hand-side is saturated and convex, it is sufficient to show $\gamma_{\mathcal{B}}(Q_1) \cup \gamma_{\mathcal{B}}(Q_2) \subseteq \gamma_{\mathcal{B}}(\dot{\min}(Q_1,Q_2))$. It follows directly from the fact that $\dot{\min}(Q_1,Q_2) \dot{\leq} Q_1$ and $\dot{\min}(Q_1,Q_2) \dot{\leq} Q_2$, as well as the definition of $\gamma_{\mathcal{B}}$.

□

A.3 Thm. 5.3

Proof. By the properties of the concrete semantics, we have $\gamma_{\mathcal{R}}(a) = \overline{\{\kappa \mid \forall s. \int y\cdot\kappa(s,dy) \leq s + a\}}$, where $\overline{C}$ is the Scott closure of C in $\mathbb{P}\Omega$.

- We want to show $\underline{1}_{\mathcal{C}} \sqsubseteq_{\mathcal{C}} \gamma_{\mathcal{R}}(\underline{1}_{\mathcal{R}})$. Appeal to the fact that $\underline{1}_{\mathbb{K}} \in \gamma_{\mathcal{R}}(0)$.

- We want to show for all $Q_1, Q_2 \in \mathcal{R}$, $\gamma_{\mathcal{R}}(Q_1) \otimes_{\mathcal{C}} \gamma_{\mathcal{R}}(Q_2) \sqsubseteq_{\mathcal{C}} \gamma_{\mathcal{R}}(Q_1 \otimes_{\mathcal{R}} Q_2)$. It is sufficient to show that $\overline{conv(\gamma_{\mathcal{R}}(Q_1) \otimes \gamma_{\mathcal{R}}(Q_2))} \subseteq \gamma_{\mathcal{R}}(Q_1 + Q_2)$. Observe that the right-hand-side is Scott-closed and convex, it is sufficient to show that $\gamma_{\mathcal{R}}(Q_1) \otimes \gamma_{\mathcal{R}}(Q_2) \subseteq \gamma_{\mathcal{R}}(Q_1 + Q_2)$. Suppose $\kappa_1 \in \gamma_{\mathcal{R}}(Q_1)$ and $\kappa_2 \in \gamma_{\mathcal{R}}(Q_2)$. Observe that $\int y \cdot \kappa_1(s, dy) \le s + Q_1$ and $\int y \cdot \kappa_2(s, dy) \le s + Q_2$. Then $\int y \cdot (\kappa_1 \otimes \kappa_2)(s, dy) = \int_y y (\int_z \kappa_1(s, dz) \cdot \kappa_2(z, dy)) = \int_z (\int_y y \cdot \kappa_2(z, dy)) \cdot \kappa_1(s, dz) \le \int (z + Q_2) \cdot \kappa_1(s, dz) = \int z \cdot \kappa_1(s, dz) + \int Q_2 \cdot \kappa_1(s, dz) \le s + Q_1 + Q_2$. Hence $\kappa_1 \otimes \kappa_2 \in \gamma_{\mathcal{R}}(Q_1 + Q_2)$.
- We want to show for all $Q_1, Q_2 \in \mathcal{R}$ and $p \in [0, 1]$, $\gamma_{\mathcal{R}}(Q_1) \, {}_p{\oplus}_{\mathcal{C}} \, \gamma_{\mathcal{R}}(Q_2) \sqsubseteq_{\mathcal{C}} \gamma_{\mathcal{R}}(Q_1 \, {}_p{\oplus}_{\mathcal{R}} \, Q_2)$. It is sufficient to show that $\overline{\gamma_{\mathcal{R}}(Q_1) \, {}_p{\oplus} \, \gamma_{\mathcal{R}}(Q_2)} \subseteq \gamma_{\mathcal{R}}(pQ_1 + (1-p)Q_2)$. Observe that the right-hand-side is Scott-closed, it is sufficient to show that $\gamma_{\mathcal{R}}(Q_1) {}_p{\oplus} \gamma_{\mathcal{R}}(Q_2) \subseteq \gamma_{\mathcal{R}}(pQ_1 + (1-p)Q_2)$. Suppose $\kappa_1 \in \gamma_{\mathcal{R}}(Q_1)$ and $\kappa_2 \in \gamma_{\mathcal{R}}(Q_2)$. Observe that $\int y \cdot \kappa_1(s, dy) \le s + Q_1$ and $\int y \cdot \kappa_2(s, dy) \le s + Q_2$. Then $\int y \cdot (\kappa_1 \, {}_p{\oplus} \, \kappa_2)(s, dy) = \int y \cdot (p\kappa_1 + (1-p)\kappa_2)(s, dy) = \int y \cdot p \cdot \kappa_1(s, dy) + \int y \cdot (1-p) \cdot \kappa_2(s, dy) \le pQ_1 + (1-p)Q_2$. Hence $\kappa_1 \, {}_p{\oplus} \, \kappa_2 \in \gamma_{\mathcal{R}}(pQ_1 + (1-p)Q_2)$.
- We want to show for all $Q_1, Q_2 \in \mathcal{R}$, $\gamma_{\mathcal{R}}(Q_1) \Cup_{\mathcal{C}} \gamma_{\mathcal{R}}(Q_2) \sqsubseteq_{\mathcal{C}} \gamma_{\mathcal{R}}(Q_1 \Cup_{\mathcal{R}} Q_2)$. It is sufficient to show that $\overline{conv(\gamma_{\mathcal{R}}(Q_1) \cup \gamma_{\mathcal{R}}(Q_2))} \subseteq \gamma_{\mathcal{R}}(\max(Q_1, Q_2))$. Observe that the right-hand-side is Scott-closed and convex, it is sufficient to show that $\gamma_{\mathcal{R}}(Q_1) \cup \gamma_{\mathcal{R}}(Q_2) \subseteq \gamma_{\mathcal{R}}(\max(Q_1, Q_2))$. It follows directly from the fact that $Q_1 \le \max(Q_1, Q_2)$ and $Q_2 \le \max(Q_1, Q_2)$, as well as the definition of $\gamma_{\mathcal{R}}$.

$\square$

A.4 Thm. 5.4

Proof. Let $\Omega \overset{\text{def}}{=} \mathbb{R}^n_{\ge 0}$ be the set of program states. (Note here we assume $\mathbb{R}_{\ge 0}$ admits a Borel-field generated by sets of compact intervals [McIver and Morgan 2001].) By the properties of the concrete semantics, we have

$$\gamma_{\mathcal{I}}(P, EP) = \overline{\{\kappa \mid \forall s. \kappa(s)((P|_s)^c) = 0 \land \begin{bmatrix} \int y \cdot \kappa(s)(dy) \\ s \end{bmatrix} \in \overline{EP}\}}.$$

- We want to show $\underline{1}_{\mathcal{C}} \sqsubseteq_{\mathcal{C}} \gamma_{\mathcal{I}}(\underline{\perp}_{\mathcal{I}})$. It is sufficient to show $\downarrow \underline{1}_{\mathbb{K}} \subseteq \gamma_{\mathcal{I}}(\{x_i' = x_i\}, \{\mathbb{E}[x_i'] = x_i\})$. Appeal to the fact that $\underline{1}_{\mathbb{K}} \in \gamma_{\mathcal{I}}(\{x_i' = x_i\}, \{\mathbb{E}[x_i'] = x_i\})$.
- We want to show for all $(P_1, EP_1), (P_2, EP_2) \in \mathcal{I}$, $\gamma_{\mathcal{I}}(P_1, EP_1) \otimes_{\mathcal{C}} \gamma_{\mathcal{I}}(P_2, EP_2) \sqsubseteq_{\mathcal{C}} \gamma_{\mathcal{I}}((P_1, EP_1) \otimes_{\mathcal{I}} (P_2, EP_2))$. It is sufficient to show that $\gamma_{\mathcal{I}}(P_1, EP_1) \otimes \gamma_{\mathcal{I}}(P_2, EP_2) \subseteq \gamma_{\mathcal{I}}(P_1 \otimes P_2, EP_1 \otimes EP_2)$. Suppose $\kappa_1 \in \gamma_{\mathcal{I}}(P_1, EP_1)$ and $\kappa_2 \in \gamma_{\mathcal{I}}(P_2, EP_2)$. Observe that $(\kappa_1 \otimes \kappa_2)(s, (P_1 \otimes P_2|_s)^c) = \int \kappa_1(s, dy) \cdot \kappa_2(y, (P_1 \otimes P_2|_s)^c)$. If $y \in (P_1|_s)^c$, then $\kappa_1(s, \{y\}) = 0$. If $y \in P_1|_s$, then by the definition of $P_1 \otimes P_2$, we know $\kappa_2(y, (P_1 \otimes P_2|_s)^c) = 0$. Hence $(\kappa_1 \otimes \kappa_2)(s, (P_1 \otimes P_2|_s)^c) = 0$. On the other hand, observe that $\int y \cdot (\kappa_1 \otimes \kappa_2)(s, dy) = \int_y y (\int_z \kappa_1(s, dz) \cdot \kappa_2(z, dy)) = \int_z (\int_y y \cdot \kappa_2(z, dy)) \cdot \kappa_1(s, dz)$, and $\begin{bmatrix} \int y \cdot \kappa_2(z, dy) \\ z \end{bmatrix} \in \overline{EP_2}$ for all z, by the fact that EP_2 is convex, we know that $\begin{bmatrix} \int_z (\int_y y \cdot \kappa_2(z, dy)) \cdot \kappa_1(s, dz) \\ \int_z z \cdot \kappa_1(s, dz) \end{bmatrix} \in \overline{EP_2}$. Because $\begin{bmatrix} \int_z z \cdot \kappa_1(s, dz) \\ s \end{bmatrix} \in \overline{EP_1}$, we know that $\begin{bmatrix} \int y \cdot (\kappa_1 \otimes \kappa_2)(s, dy) \\ y \end{bmatrix} \in \overline{EP_1} \otimes \overline{EP_2} \subseteq \overline{EP_1 \otimes EP_2}$.
- We want to show for all $(P_1, EP_1), (P_2, EP_2) \in \mathcal{I}$ and $\varphi \in \mathcal{L}$, $\gamma_{\mathcal{I}}(P_1, EP_1) \, {}_{\varphi}{\diamond}_{\mathcal{C}} \, \gamma_{\mathcal{I}}(P_2, EP_2) \sqsubseteq_{\mathcal{C}} \gamma_{\mathcal{I}}((P_1, EP_1) \, {}_{\varphi}{\diamond}_{\mathcal{I}} \, (P_2, EP_2))$. It is sufficient to show that $\gamma_{\mathcal{I}}(P_1, EP_1) \, {}_{\varphi}{\diamond} \, \gamma_{\mathcal{I}}(P_2, EP_2) \subseteq \gamma_{\mathcal{I}}(P \overset{\text{def}}{=} (\{\varphi\} \sqcap P_1) \sqcup (\{\neg\varphi\} \sqcap P_2), (EP_1 \sqcup EP_2) \sqcap \overline{P[\mathbb{E}[x']/x']})$. Suppose $\kappa_1 \in \gamma_{\mathcal{I}}(P_1, EP_1)$ and $\kappa_2 \in \gamma_{\mathcal{I}}(P_2, EP_2)$. Observe that $(\kappa_1 \, {}_{\varphi}{\diamond} \, \kappa_2)(s, (P|_s)^c) = \textbf{if } \widehat{\varphi}(s) \textbf{ then } \kappa_1(s, (P|_s)^c) \textbf{ else } \kappa_2(s, (P|_s)^c)$. If $\widehat{\varphi}(s)$, then

$P|_s = \{\varphi\} \sqcap P_1 \subseteq P_1$, hence $(\kappa_1 \, {}_{\varphi}\!\diamond \, \kappa_2)(s, (P|_s)^c) = 0$. If $\neg\widehat{\varphi}(s)$, then $P|_s = \{\neg\varphi\} \sqcap P_2 \subseteq P_2$, hence $(\kappa_1 \, {}_{\varphi}\!\diamond \, \kappa_2)(s, (P|_s)^c) = 0$. On the other hand, observe that $\int y \cdot (\kappa_1 \, {}_{\varphi}\!\diamond \, \kappa_2)(s, dy) = \textbf{if } \widehat{\varphi}(s) \textbf{ then } \int y \cdot \kappa_1(s, dy) \textbf{ else } \int y \cdot \kappa_2(s, dy)$. Hence $\left[\begin{array}{c} \int y \cdot (\kappa_1 \, {}_{\varphi}\!\diamond \, \kappa_2)(s, dy) \\ s \end{array} \right] \in \overline{EP_1} \sqcup \overline{EP_2} \subseteq \overline{EP_1 \sqcup EP_2}$.

- We want to show for all $(P_1, EP_1), (P_2, EP_2) \in \mathcal{I}$ and $p \in [0,1]$, $\gamma_{\mathcal{I}}(P_1, EP_1) \, {}_{p}\!\oplus_{\mathcal{C}} \, \gamma_{\mathcal{I}}(P_2, EP_2) \sqsubseteq_{\mathcal{C}} \gamma_{\mathcal{I}}((P_1, EP_1) \, {}_{p}\!\oplus_{\mathcal{I}} \, (P_2, EP_2))$. It is sufficient to show that $\gamma_{\mathcal{I}}(P_1, EP_1) \, {}_{p}\!\oplus \, \gamma_{\mathcal{I}}(P_2, EP_2) \subseteq \gamma_{\mathcal{I}}(P_1 \sqcup P_2, EP_1 \, {}_{p}\!\oplus \, EP_2)$. Suppose $\kappa_1 \in \gamma_{\mathcal{I}}(P_1, EP_1)$ and $\kappa_2 \in \gamma_{\mathcal{I}}(P_2, EP_2)$. Observe that $(\kappa_1 \, {}_{p}\!\oplus \, \kappa_2)(s, (P_1 \sqcup P_2|_s)^c) = p \cdot \kappa_1(s, (P_1 \sqcup P_2|_s)^c) + (1-p) \cdot \kappa_2(s, (P_1 \sqcup P_2|_s)^c) = 0$. On the other hand, observe that $\int y \cdot (\kappa_1 \, {}_{p}\!\oplus \, \kappa_2)(s, dy) = p \int y \cdot \kappa_1(s, dy) + (1-p) \int y \cdot \kappa_2(s, dy)$. Hence $\kappa_1 \, {}_{p}\!\oplus \, \kappa_2 \in \overline{EP_1 \, {}_{p}\!\oplus \, EP_2}$.
- We want to show that for all $(P_1, EP_1), (P_2, EP_2) \in \mathcal{I}$, $\gamma_{\mathcal{I}}(P_1, EP_1) \Cup_{\mathcal{C}} \gamma_{\mathcal{I}}(P_2, EP_2) \sqsubseteq_{\mathcal{C}} \gamma_{\mathcal{I}}((P_1, EP_1) \Cup_{\mathcal{I}} (P_2, EP_2))$. Appeal to the fact that $\gamma_{\mathcal{I}}(P_1, EP_1) \cup \gamma_{\mathcal{I}}(P_2, EP_2) \subseteq \gamma_{\mathcal{I}}(P_1 \sqcup P_2, EP_1 \sqcup EP_2)$.

□