# Refuting Equivalence in Probabilistic Programs with Conditioning

Krishnendu Chatterjee[1] , Ehsan Kafshdar Goharshady[1(✉)] ,
Petr Novotný[2] , and Đorđe Žikelić[3]

[1] Institute of Science and Technology Austria (ISTA), Klosterneuburg, Austria
`{krishnendu.chatterjee,ehsan.goharshady}@ist.ac.at`
[2] Masaryk University, Brno, Czech Republic
`petr.novotny@fi.muni.cz`
[3] Singapore Management University, Singapore, Singapore
`dzikelic@smu.edu.sg`

**Abstract.** We consider the problem of refuting equivalence of probabilistic programs, i.e., the problem of proving that two probabilistic programs induce different output distributions. We study this problem in the context of programs with conditioning (i.e., with observe and score statements), where the output distribution is conditioned by the event that all the observe statements along a run evaluate to true, and where the probability densities of different runs may be updated via the score statements. Building on a recent work on programs without conditioning, we present a new equivalence refutation method for programs with conditioning. Our method is based on weighted restarting, a novel transformation of probabilistic programs with conditioning to the output equivalent probabilistic programs without conditioning that we introduce in this work. Our method is the first to be both a) fully automated, and b) providing provably correct answers. We demonstrate the applicability of our method on a set of programs from the probabilistic inference literature.

## 1 Introduction

*Probabilistic programs.* Probabilistic programs (PPs) are standard programs extended with *probabilistic instructions*, such as sampling of variable values from probability distributions. PPs are used in numerous application domains, such as networking [33], privacy and security [7,8], or planning and robotics [35,59], to name a few. Unlike deterministic programs, which (assuming they terminate) for each concrete input produce a single output, PPs induce *output distributions*, i.e. *probability distributions* over the space of possible outputs.

*Conditioning.* While PPs with sampling instructions can implement various randomized algorithms, the recent proliferation of probabilistic programming is driven largely by the advances in machine learning, where PPs serve as tools for Bayesian inference. For this, a PP has to be equipped with observe statements which can be used to condition the data in program's variables through

observations, as well as `score` statements which can be used to assign different probability densities to different program executions. Probability of the non-blocked runs is then renormalized [43,53,12]. In this way, the program's output distribution is *conditioned* by the event that the program's run satisfies all observe-statement assertions it encounters and all program run probability densities are *renormalized* according to their cummulative score values.

*Static and relational analysis of PPs.* Given their inherent randomness, PPs are extremely difficult to debug. This has spurred demand for formal analyses of PPs so as to obtain provable assurances of their correctness. Over the past decade, numerous static analyses for PPs were developed, focusing mostly on properties of *individual programs:* termination (e.g. [47,14,32,19,25,1,49,26,22,57,58,46]), safety [55,25,57,20,11,10,64], runtime and cost analysis [52,63,5,61,29,21], or input-output behavior [27]. Recently, attention has been increasingly given to verifying *relational properties* of PPs, i.e., properties of program *pairs,* such as sensitivity [6,42,62,2] or differential privacy [7,3].

*Refuting equivalence of PPs.* In this work, we study the relational property of program *equivalence.* Two PPs with fixed inputs are called equivalent if they induce the same output distributions [51,45,48]. Equivalence analysis allows one to check, e.g., whether compiler optimizations preserve the input-output behavior of the program, or if a sampler follows a desired distribution [15]. We focus on a bug-hunting aspect of the equivalence property, i.e., *equivalence refutation:* given two PPs, prove that their output distributions differ. Recently, the work [23] presented an automated method for equivalence refutation in PPs that has formal correctness guarantees. However, the method of [23] works only for PPs *without conditioning.* In this paper, we focus on refuting equivalence of PPs *with conditioning*, i.e., proving the inequality of their *conditional output distributions.*

**Our contributions.** We present a new method for refuting equivalence of PPs. Our method is the first to have *all* of the following properties: (i) It is fully automated, (ii) it is applicable infinite-state, discrete and continuous, Turing-complete PPs *with conditioning*, and (iii) it provides provably correct refutations.

We build on the notions of *upper and lower expectation supermartingales (U/LESMs)* presented in [23]. U/LESMs map program states to real numbers, providing upper and lower bounds on the expected value of some function $f$ *at output.* During the refutation process, the concrete function $f$ is synthesized along with the corresponding UESM and LESM to provide the required refutation proof: for a given pair $(P_1, P_2)$ of PPs, we seek $f$ whose expected value at the output of $P_1$ is provably different (due to the bounds provided by U/LESMs) from its expected value at the output of $P_2$ - this witnesses the difference of the underlying output distributions. The computation of $f$ and the U/LESMs proceeds via polynomial constraint solving. However, the method is restricted to PPs without conditioning, and its extension to PPs with conditioning was left as an open problem (see [23, Section 9]). Our contributions are as follows:

1. *Equivalence refutation for PPs with conditioning.* We extend the above to the much more general setting of PPs with conditioning, allowing for both `observe` and `score` statements. We achieve this via the notion of *weighted*

*restarting in PPs* that we introduce in this work. Weighted restarting is a construction that translates a PP with conditioning into a PP without conditioning, whose output distribution is equivalent to that of the original PP. Hence, by applying weighted restarting to two PPs with conditioning, we can reduce their equivalence refutation to refuting equivalence of a pair of PPs without conditioning, for which we can utilize the method of [23].

2. *Soundness and completeness guarantees.* While the method of [23] was proved to be sound, completeness of the proof rule was not studied and, to the best of our knowledge, no sound and complete proof rule for equivalence refutation in PPs has been proposed. In this work, we show that the proof rule based on U/LESMs is *sound and complete* for refuting equivalence. This result is novel both in the settings of PPs with and without conditioning. Moreover, it also allows us to prove that the polynomial constraint solving algorithm for computing a function $f$ over program outputs together with the U/LESM is *semi-complete*, in the sense that it is guaranteed to be able to refute equivalence of PPs whenever equivalence can be witnessed by a function $f$ and a pair of U/LESM that can be specified via polynomial functions of some maximal polynomial degree $D$. Note that this is a significant guarantee, given that the problem of equivalence checking of two programs is undecidable [36], hence no algorithm can be both sound and complete.

3. *Experimental evaluation.* We evaluate our approach on benchmarks from the well known PSI inference tool [34], as well as on several loopy programs. We compare with a PSI+Mathematica [66] pipeline. Our method significantly outperforms the baseline in terms of the number of refuted equivalences.

Finally, while the focus of this work is on equivalence refutation, weighted restarting also allows us to design a new method for *similarity refutation* in PPs with conditioning. The similarity refutation problem is concerned with computing a lower bound on the distance between output distributions of two PPs. The work [23] proposed a method for similarity refutation in PPs without conditioning, and in this work we generalize it to the much more general setting of PPs with conditioning. Due to space restrictions, we defer this result to the extended version of the paper [24]. However, in our experiments in Section 6, for each refuted equivalence we also report a lower bound on the Kantorovich distance [60] computed by our method.

*Related work.* The study of PP equivalence is partly motivated by the necessity to reason about the correctness and accuracy of probabilistic inference algorithms. Previous approaches to this issue [9,16,39,38,28,13,30] employ sampling methods that provide statistical guarantees on the correctness of the result.

Relational analyses of PPs mostly focused on *sensitivity properties* [6,42,62,2], often using *coupling proofs* [7,3]. Intuitively, given a probabilistic program and its two inputs, the goal here is to bound the distance of output distributions corresponding to these inputs. In particular, the relational analysis in this context focuses on relations between different *inputs,* rather than different *programs.* Indeed, the previous approaches to sensitivity analysis typically exploit the "aligned" control flow of two executions on sufficiently close inputs (with

a notable exception of the recent work [37] which aims to dispel the alignment requirement.) Our method makes no such assumptions. Moreover, none of the aforementioned works except for [42] considers programs with conditioning.

The work [67] uses "potential functions" to analyze resource usage difference of a program pair. Potential functions for the two programs are computed simultaneously, similarly to our approach. However, the work does not involve probabilistic programs and its overall aim (cost analysis) differs from ours.

## 2 Preliminaries

We assume that the reader is familiar with notions from probability theory such as probability measure, random variable and expected value. The term *probability distribution* refers to a probability measure over a subset of Euclidean space $\mathbb{R}^n$. We use boldface notation to represent vectors, e.g. $\mathbf{x}$ or $\mathbf{y}$.

*Syntax.* We consider imperative arithmetic probabilistic programs (PPs) allowing standard programming constructs such as variable assignments, if-branching, sequential composition, go-to statements, and loops. All program variables are assumed to be integer- or real-valued. For PP semantics to be well defined, we assume that all arithmetic expressions are Borel-measurable functions [65].

In addition, we allow two types of probabilistic instructions, namely *sampling* and *conditioning*. Sampling instructions appear on the right-hand-side of program variable assignments and are represented in our syntax by a command **sample**($d$), where $d$ is a probability distribution. We do not impose specific restrictions and allow sampling instructions from both discrete and continuous distributions. Importantly, we allow the two standard conditioning instructions: **observe**($\phi$), where $\phi$ is a boolean predicate over program variables, and **score**($e$), where $e$ is a non-negative expression over program variables. A program is *conditioning-free* if it does not contain any conditioning instructions.

Probabilistic branching instructions, i.e. **if prob**($p$), can be obtained as syntactic sugar by using a sampling instruction followed by conditional branching.

*Probabilistic control flow graphs (pCFGs).* In order to formalize the semantics of our PPs and to present our results and algorithms, we consider an operational representation of PPs called *probabilistic control-flow graphs (pCFGs)* extended with the concept of *weighting* as in [64]. The use of pCFGs in the analysis of PPs is standard, see e.g. [19,1]. Hence, we keep this exposition brief. Formally, a pCFG is a tuple $\mathcal{C} = (L, V, V_{out}, \ell_{in}, \mathbf{x}_{in}, \mapsto, G, Up, w, \ell_{out})$, where:

- $L$ is a finite set of *locations*;
- $V = \{x_1, \ldots, x_{|V|}\}$ is a finite set of *program variables*;
- $V_{out} = \{x_1, \ldots, x_{|V_{out}|}\} \subseteq V$ is a finite set of *output variables*;
- $\ell_{in} \in L$ is the *initial location* and $\mathbf{x}_{in} \in \mathbb{R}^{|V|}$ the *initial variable valuation*;
- $\mapsto \subseteq L \times L$ is a finite set of *transitions*;
- $G$ is a map assigning to each transition $\tau \in \mapsto$ a *guard* $G(\tau)$, which is a predicate over $V$ specifying whether $\tau$ can be executed.
- $Up$ is a map assigning to each transition $\tau \in \mapsto$ an *update* $Up(\tau) = (u_1, \ldots, u_{|V|})$ where for each $j \in \{1, \ldots, |V|\}$ the variable update $u_j$ is either

$\ell_{in}:$  $x, y, c, r = 0$
$\ell_1:$  **while**  $c \leq 999:$
$\ell_2:$      **if**  **prob**$(0.5):$
$\ell_3:$          $r := \mathbf{Beta}(2, 2)$
$\ell_4:$          $x := x - r$
      **else**:
$\ell_5:$          $y := y + 1$
$\ell_6:$      $c = c + 1$
$\ell_7:$    **score**$(y)$
$\ell_{out}:$  **return**$(x, y)$

$\ell_{in}:$  $x, y, c, r = 0$
$\ell_1:$  **while**  $c \leq 999:$
$\ell_2:$      **if**  **prob**$(0.5):$
$\ell_3:$          $r := \mathbf{Beta}(2, 3)$
$\ell_4:$          $x := x - r$
      **else**:
$\ell_5:$          $y := y + 1$
$\ell_6:$      $c = c + 1$
$\ell_7:$    **score**$(y)$
$\ell_{out}:$  **return**$(x, y)$

Fig. 1: Running example. Labels $\ell_{in}, \ell_1, \ldots, \ell_7, \ell_{out}$ correspond to locations in the probabilistic control-flow graphs (pCFGs) of both programs.

- the bottom element $\perp$ denoting no variable update, or
- a Borel-measurable arithmetic expression $u : \mathbb{R}^{|V|} \to \mathbb{R}$, or
- a probability distribution $u = \delta$;
- $w : \mapsto \times \mathbb{R}^{|V|} \to [0, \infty)$ is a *weighting function*, and
- $\ell_{out} \in L$ the *terminal location*.

Translation of PPs into pCFGs is standard [19,1,64], hence we omit the details. We denote by $\mathcal{C}(P)$ the pCFG induced by a program $P$. Intuitively, locations of $\mathcal{C}(P)$ correspond to program instructions, while transition(s) out of a location encodes the effects of the respective instruction. Weight of a transition is 1 unless the transition encodes one of the conditioning instructions: If $\tau$ is encodes a **score**$(e)$ instruction, we put $w(\tau, \mathbf{x}) = e(\mathbf{x})$. If $\tau$ encodes an **observe**$(\phi)$ statement, we put $w(\tau, \mathbf{x}) = 1$ if $\mathbf{x} \models \phi$ and $w(\tau, \mathbf{x}) = 0$ otherwise. Note that if $P$ is conditioning-free then in $\mathcal{C}(P)$ we have $w(\tau, \mathbf{x}) = 1$ for all $\tau$ and $\boldsymbol{x}$.

*Example 1 (Running example).* Fig. 1 shows a PP pair that will serve as our running example. Program variables in both PPs are $V = \{x, y, c, r\}$, with output variables $V_{out} = \{x, y\}$. In each loop iteration, with probability 0.5 the variable $x$ is decremented by a random value sampled according to a beta distribution, whereas with probability 0.5 the variable $y$ is incremented by 1. Hence, $y$ tracks the number of loop iterations in which $x$ is not decremented. Upon loop termination, the **score** command is used to multiply the probability density of each run by the value of $y$ upon termination. Thus, larger probability density is given to program runs that modify $x$ a smaller number of times. The programs differ only in the parameters of the beta distribution (highlighted in blue).

*States, paths and runs.* A *state* in a pCFG $\mathcal{C}$ is a tuple $(\ell, \mathbf{x})$, where $\ell$ is a location and $\mathbf{x} \in \mathbb{R}^{|V|}$ is a variable valuation. A transition $\tau = (\ell, \ell')$ is *enabled* at a state $(\ell, \mathbf{x})$ if $\mathbf{x} \models G(\tau)$. A state $(\ell', \mathbf{x}')$ is a *successor* of $(\ell, \mathbf{x})$, if there exists an enabled transition $\tau = (\ell, \ell')$ in $\mathcal{C}$ such that $\mathbf{x}'$ is a possible result of applying the update of $\tau$ to $\mathbf{x}$. The state $(\ell_{in}, \mathbf{x}_{in})$ is the *initial state*. A state $(\ell, \mathbf{x})$ is said to be *terminal*, if $\ell = \ell_{out}$. We use $State^{\mathcal{C}}$ to denote the set of all states in $\mathcal{C}$.

A *finite path* in $\mathcal{C}$ is a sequence of successor states $(\ell_0, \mathbf{x}_0), (\ell_1, \mathbf{x}_1), \ldots, (\ell_k, \mathbf{x}_k)$ with $(\ell_0, \mathbf{x}_0) = (\ell_{in}, \mathbf{x}_{in})$. A state $(\ell, \mathbf{x})$ is *reachable* in $\mathcal{C}$ if there exists a finite

path in $\mathcal{C}$ whose last state is $(\ell, \mathbf{x})$. A *run* (or an *execution*) in $\mathcal{C}$ is an infinite sequence of states whose each finite prefix is a finite path. We use $Fpath^{\mathcal{C}}$, $Run^{\mathcal{C}}$, and $Reach^{\mathcal{C}}$ to denote the sets of all finite paths, runs, and reachable states in $\mathcal{C}$.

A run $\rho \in Run^{\mathcal{C}}$ is *terminating* if it reaches some terminal state $(\ell_{out}, \mathbf{x})$. We use $Term \subseteq Run^{\mathcal{C}}$ to denote the set of terminating runs in $Run^{\mathcal{C}}$. We define the *termination time* of $\rho$ via $T(\rho) = \inf_{i \geq 0}\{i \mid \ell_i = \ell_{out}\}$, with $T(\rho) = \infty$ if $\rho$ is not terminating. A *cumulative weight* of a terminating run $\rho = (\ell_0, \mathbf{x}_0), (\ell_1, \mathbf{x}_1), \ldots$ is $W(\rho) = \prod_{i=0}^{T(\rho)-1} w((\ell_i, \ell_{i+1}), \mathbf{x}_i)$, with an empty product being equal to 1.

*Semantics.* The pCFG semantics are formalized as Markov chains with weights [64]. In particular, a pCFG $\mathcal{C}$ defines a discrete-time Markov process over the states of $\mathcal{C}$ whose trajectories correspond to runs in $\mathcal{C}$. Each trajectory starts in the initial state $(\ell_{in}, \mathbf{x}_{in})$ with weight 1. Then, at each time step $i \in \mathbb{N}_0$, if the trajectory is in state $(\ell_i, \mathbf{x}_i)$, the next trajectory state $(\ell_{i+1}, \mathbf{x}_{i+1})$ is defined according to the unique pCFG transition $\tau_i$ enabled at $(\ell_i, \mathbf{x}_i)$, and the weight of the trajectory is multiplied by the value of the weight function $w(\tau_i, \mathbf{x}_i)$.

This Markov process gives rise to a probability space $(Run^{\mathcal{C}}, \mathcal{F}^{\mathcal{C}}, \mathbb{P}^{\mathcal{C}})$ over the set of all pCFG runs, formally defined via the cylinder construction [50]. We use $\mathbb{E}^{\mathcal{C}}$ to denote the expectation operator in this probability space.

*Almost-sure termination.* We restrict our attention to almost-surely terminating programs, which is necessary for them to define valid probability distributions on output. A program $P$ (or, equivalently, its pCFG $\mathcal{C}(P)$) terminates *almost-surely (a.s.)* if $\mathbb{P}^{\mathcal{C}(P)}[Term] = 1$. A.s. termination can be automatically verified in arithmetic PPs, by e.g. synthesizing a ranking supermartingale (RSM) [14,17].

## 3    Equivalence Refutation Problem

For each variable valuation $\mathbf{x} \in \mathbb{R}^{|V|}$, we use $\mathbf{x}^{out}$ to denote its projection to the components of output variables $V_{out}$. For a terminating run $\rho$ that reaches a terminal state $(\ell_{out}, \mathbf{x})$, we say that $\mathbf{x}^{out}$ is the *output* of $\rho$. In an a.s. terminating pCFG $\mathcal{C}$, the probability measure $\mathbb{P}^{\mathcal{C}}$ over runs naturally extends to a probability measure over sets of outputs, weighted by the cumulative weights of individual runs and normalized by the total expected weight over all runs. Formally, the *normalized output distribution (NOD)* of $\mathcal{C}$ is the probability distribution over all output variable valuations $\mathbb{R}^{|V_{out}|}$ defined by

$$\mu^{\mathcal{C}}[B] = \frac{\mathbb{E}^{\mathcal{C}}\big[\mathbb{I}_{Output(B)} \cdot W\big]}{\mathbb{E}^{\mathcal{C}}\big[W\big]},$$

where (i) $B$ is any Borel-measurable subset of $\mathbb{R}^{|V_{out}|}$, (ii) $\mathbb{I}_A$ is the indicator variable of a Borel-measurable set of runs $A$, and (iii):

$$Output(B) = \big\{\rho \in Run^{\mathcal{C}} \mid \rho \text{ reaches a terminal state } (\ell_{out}, \mathbf{x}) \text{ s.t. } \mathbf{x}^{out} \in B\big\}.$$

For the NOD $\mu^{\mathcal{C}}$ to be well-defined, we restrict our attention to pCFGs where $0 < \mathbb{E}^{\mathcal{C}}\big[W\big] < \infty$, in which case we call $\mathcal{C}$ *integrable* (in line with [64]).

*Problem assumptions.* For a pair of PPs $P^1$ and $P^2$, we assume that: (1) both $\mathcal{C}(P^1)$ and $\mathcal{C}(P^2)$ are a.s. terminating and integrable, for their output distributions to be well defined, and (2) $\mathcal{C}(P^1)$ and $\mathcal{C}(P^2)$ share the same set of output variables $V_{out}$, for their output distributions to be defined over the same sample space so that their equivalence is well defined. Our algorithm will automatically check (1), and (2) is trivially checked by comparing the two input pCFGs.

**Problem.** Let $P^1$ and $P^2$ be two PPs with pCFGs $\mathcal{C}(P^1)$ and $\mathcal{C}(P^2)$ being a.s. terminating, integrable, and having the same set of output variables $V_{out}$. Our goal is to prove that $P^1$ and $P^2$ are *not (output) equivalent*, i.e. that there exists a Borel-measurable set $B \subseteq \mathbb{R}^{|V_{out}|}$ such that $\mu^{\mathcal{C}(P^1)}[B] \neq \mu^{\mathcal{C}(P^2)}[B]$.

## 4    Equivalence Refutation Proof Rule

In this section, we first introduce the weighted restarting transformation, and then use it to formulate a proof rule for equivalence refutation in programs with conditioning. We also show the completeness of this proof rule.

### 4.1    Weighted Restarting

We introduce weighted restarting, which transforms a PP into output equivalent conditioning-free PP. Our method of weighted restarting requires PPs and their induced pCFGs to satisfy the bounded cumulative weight property.

*Assumption: Bounded cumulative weight property.* We say that a program $P$ (or equivalently, its pCFG $\mathcal{C}(P)$) satisfies the *bounded cumulative weight property*, if there exists $M > 0$ such that the cumulative weight of every terminating run $\rho$ is bounded from above by $M$, i.e. $W(\rho) \leq M$. Note that, in an a.s. terminating pCFG $\mathcal{C}$, the bounded cummulative weight property also implies the upper bound in the integrability property, since if $W(\rho) \leq M$ holds for all terminating runs then we also have $\mathbb{E}^{\mathcal{C}(P)}[W] \leq M < \infty$. Our algorithm for equivalence refutation in Section 5 will formally verify this property and compute the value of $M$.

*Weighted restarting.* Consider a PP $P$ which is a.s. terminating, integrable, and satisfies the bounded cummulative weight property with bound $M > 0$. Weighted restarting produces an output equivalent conditioning-free PP $P_{\text{res}}$. To achieve this, it introduces a fresh program variable $W$, which we call the *weight variable*, to capture all information encoded via observe and score instructions. This allows us to remove all conditioning instructions from the PP. Then, for the output distribution to be equivalent to the *normalized output distribution* of the original PP, it adds a block of code that *terminates* a program run with probability proportional to its cumulative weight, and otherwise *restarts* it and moves it back to the initial program state. This construction is formalized as follows:

1. *Introduce weight variable $W$.* Introduce a fresh program variable $W$. The value of $W$ is initially set to $W = 1$.

2. *Remove observe instructions.* Each **observe**($\phi$) instruction is replaced by the following block of code, which sets the cumulative weight of a run to 0 if the predicate in the observe instruction is false:

$$\textbf{if } \neg\phi :$$
$$W := 0$$

3. *Remove score instructions.* Each **score**($e$) instruction is replaced by the following block of code, which multiplies the weight variable $W$ by the value of the expression in the score instruction:

$$W := W \cdot e$$

4. *Add restarting upon termination.* Finally, the following block of code is added at the end of the PP, which with probability $1 - W/M$ "restarts" a program run by moving it back to the initial state and resetting its weight to $W = 1$. We write "$\mathbf{x} := \mathbf{x}_{in}$" as a shorthand for the block of code that restarts each program variable value to that specified by the initial variable valuation $\mathbf{x}_{in}$:

$$\textbf{if } \textbf{prob}(1 - W/M) :$$
$$\mathbf{x} := \mathbf{x}_{in}$$
$$W := 1$$
$$\textbf{go-to } \ell_{in}$$

The work [54] also proposed a "restarting" procedure for PPs with observe, but without score, instructions. This is achieved by introducing a new boolean variable `unblocked` whose value is initially true, but is set to false if any **observe**($\phi$) condition is violated. The program is then embedded into a "**while unblocked:**" loop. However, it is not clear how to extend this translation to PPs with score instructions, whereas our method supports both observe and score instructions.

**Theorem 1 (Correctness of weighted restarting, proof in [24]).** *Consider a PP $P$ which is a.s. terminating, integrable, and satisfies the bounded cumulative weight property with bound $M > 0$. Let $P_{res}$ be a conditioning-free PP produced by the weighted restarting procedure. Then, $P_{res}$ is also a.s. terminating, integrable, and satisfies the bounded cumulative weight property. Moreover, $\mathcal{C}(P)$ and $\mathcal{C}(P_{res})$ are output equivalent, i.e. $\mu^{\mathcal{C}(P)} = \mu^{\mathcal{C}(P_{res})}$.*

*Example 2.* Fig. 2 shows the PPs produced by the weighted restarting procedure applied to PPs in Fig. 1. Notice that the PPs in Fig. 1 are a.s. terminating, integrable, and satisfy the bounded cumulative weight property with $M = 1000$, as $y \leq 1000$ holds upon loop termination. Weighted restarting produces the PPs in Fig. 2. In the fourth step of the procedure, the restarting upon termination block of code is added to each program in lines $7 - 10$. It can be realized using a single pCFG transition, hence we only need one pCFG location $\ell_8$ for it.

$\ell_{in}:$  $x, y, c, r = 0, W = 0$
$\ell_1:$  **while**  $c \leq 999:$
$\ell_2:$      **if**  **prob**$(0.5):$
$\ell_3:$          $r := \mathbf{Beta}(2, 2)$
$\ell_4:$          $x := x - r$
        **else**:
$\ell_5:$          $y := y + 1$
$\ell_6:$      $c = c + 1$
$\ell_7:$  $W := W \cdot y$
$\ell_8:$  **if prob**$(1 - W/1000):$
          $x, y, c, r := 0$
          $W := 1$
          **go-to** $\ell_{in}$
$\ell_{out}:$  **return** $(x, y)$

$\ell_{in}:$  $x, y, c, r = 0, W = 0$
$\ell_1:$  **while**  $c \leq 999:$
$\ell_2:$      **if**  **prob**$(0.5):$
$\ell_3:$          $r := \mathbf{Beta}(2, 3)$
$\ell_4:$          $x := x - r$
        **else**:
$\ell_5:$          $y := y + 1$
$\ell_6:$      $c = c + 1$
$\ell_7:$  $W := W \cdot y$
$\ell_8:$  **if prob**$(1 - W/1000):$
          $x, y, c, r := 0$
          $W := 1$
          **go-to** $\ell_{in}$
$\ell_{out}:$  **return** $(x, y)$

Fig. 2: PPs produced by weighted restarting applied to the PPs in Fig. 1. The changes introduced by the weighted restarting procedure are highlighted in red.

## 4.2   Equivalence Refutation in PPs with Conditioning

Theorem 1 effectively reduces the task of equivalence refutation for a PP pair $(P^1, P^2)$ into the respective task for $(P^1_{\text{res}}, P^2_{\text{res}})$ of PPs without conditioning. Following such a reduction, we can apply the refutation rule from [23]. We briefly revisit this proof rules. We additionally show its completeness in settings both with and without conditioning, a result not proven previously.

In what follows, let $\mathcal{C} = (L, V, V_{out}, \ell_{in}, \mathbf{x}_{in}, \mapsto, G, Up, w, \ell_{out})$ be an a.s. terminating pCFG. We use the following terminology:

- A *state function* $\eta$ in $\mathcal{C}$ is a function which to each location $\ell \in L$ assigns a Borel-measurable function $\eta(\ell) : \mathbb{R}^{|V|} \to \mathbb{R}$ over program variables.
- A *predicate function* $\Pi$ in $\mathcal{C}$ is a function which to each location $\ell \in L$ assigns a predicate $\Pi(\ell)$ over program variables. It naturally induces a set of states $\{(\ell, \mathbf{x}) \mid \mathbf{x} \models \Pi(\ell)\}$, which we also refer to via $\Pi$. $\Pi$ is an *invariant* if it contains all reachable states in $\mathcal{C}$, i.e. if $\mathbf{x} \models I(\ell)$ for each $(\ell, \mathbf{x}) \in Reach^{\mathcal{C}}$.

Let $f : \mathbb{R}^{|V_{out}|} \to \mathbb{R}$ be a Borel-measurable function over the pCFG outputs. A UESM (resp. LESM) for $f$ is a state function $U_f$ (resp. $L_f$) that satisfies certain conditions in every reachable state which together ensure that it evaluates to an upper (resp. lower) bound on the expected value of $f$ on the pCFG output. Since it is generally not feasible to exactly compute the set of reachable states, UESMs and LESMs are defined with respect to a supporting invariant that over-approximates the set of all reachable states in the pCFG. This is done in order to later allow for a fully automated computation of UESMs and LESMs in PPs.

**Definition 1 (Upper expectation supermartingale (UESM) [23]).** *Let $\mathcal{C}$ be an a.s. terminating pCFG, $I$ be an invariant in $\mathcal{C}$ and $f : \mathbb{R}^{|V_{out}|} \to \mathbb{R}$ be a Borel-measurable function over the output variables of $\mathcal{C}$. An* upper expectation supermartingale (UESM) *for $f$ with respect to the invariant $I$ is a state function $U_f$ satisfying the following two conditions:*

1. Zero on output. *For every* $\mathbf{x} \models I(\ell_{out})$, *we have* $U_f(\ell_{out}, \mathbf{x}) = 0$.
2. Expected $f$-decrease. *For every location* $\ell \in L$, *transition* $\tau = (\ell, \ell') \in \mapsto$, *and variable valuation* $\mathbf{x}$ *s.t.* $\mathbf{x} \models I(\ell) \wedge G(\tau)$, *we require the following: denoting by* $\mathbf{N}$ *the expected valuation vector after performing transition* $\tau$ *from state* $(\ell, \boldsymbol{x})$, *it holds*

$$U_f(\ell, \mathbf{x}) + f(\mathbf{x}^{out}) \geq \mathbb{E}\Big[U_f(\ell', \mathbf{N}) + f(\mathbf{N}^{out})\Big] \tag{1}$$

*with* $\mathbf{N}^{out}$ *the projection of the random vector* $\mathbf{N}$ *onto* $V_{out}$-*indexed variables.*

Intuitively, an UESM is required to be equal to 0 on output and, in every computation step, any increase in the $f$-value is *exceeded in expectation* by the decrease in the UESM value. A *lower expectation submartingale* (LESM) $L_f$ for $f$ is defined analogously, with (1) replaced by the dual *expected $f$-increase* condition:

$$L_f(\ell, \mathbf{x}) + f(\mathbf{x}^{out}) \leq \mathbb{E}\Big[L_f(\ell', \mathbf{N}) + f(\mathbf{N}^{out})\Big]. \tag{2}$$

*Example 3.* Consider the two PPs obtained by weighted restarting in Fig. 2. Their output variables are $V_{out} = \{x, y\}$. By Theorem 1, both PPs are a.s. terminating. For a function over outputs $f(x, y) = x + y$, one can check by inspection that the state function $U_f$ below defines a UESM for $f$ in the first program, and the state function $L_f$ below defines an LESM for $f$ in the second program. These are also the U/LESMs produced by our prototype implementation:

$$U_f\begin{pmatrix} x, \\ y, \\ c, \\ r, \\ W \end{pmatrix} = \begin{cases} 500 - 0.5 \cdot y, & \text{if } \ell \in \{\ell_{in}, \ell_1, \ell_2, \ell_6, \ell_7, \ell_8\} \\ 499.5 - 0.5 \cdot y, & \text{if } \ell = \ell_3 \\ 500 - 0.5 \cdot y - r, & \text{if } \ell = \ell_4 \\ 500.5 - 0.5 \cdot y, & \text{if } \ell = \ell_5 \\ 0, & \text{if } \ell = \ell_{out} \end{cases} \qquad L_f\begin{pmatrix} x, \\ y, \\ c, \\ r, \\ W \end{pmatrix} = \begin{cases} 600 - 0.6 \cdot y, & \text{if } \ell \in \{\ell_{in}, \ell_1, \ell_2, \ell_6, \ell_7, \ell_8\} \\ 599.6 - 0.6 \cdot y, & \text{if } \ell = \ell_3 \\ 600 - 0.6 \cdot y - r, & \text{if } \ell = \ell_4 \\ 600.4 - 0.6 \cdot y, & \text{if } \ell = \ell_5 \\ 0, & \text{if } \ell = \ell_{out} \end{cases}$$

U/LESMs can be used to refute equivalence in conditioning-free PPs under so-called *OST-soundness conditions*. The name is due to the fact that correctness of the proof rule in [23] is proven via Optional Stopping Theorem (OST) [65] from martingale theory. The first three conditions below are imposed by the classical OST, and the fourth condition is derived from the Extended OST [63].

**Definition 2 (OST-soundness).** *Let* $\mathcal{C}$ *be a pCFG,* $\eta$ *be a state function in* $\mathcal{C}$, *and* $f \colon \mathbb{R}^{|V_{out}|} \to \mathbb{R}$ *be a Borel measurable function. Let* $S_i$ *be the $i$-th state and* $\mathbf{x}_i$ *the $i$-th variable valuation along a run. Define* $Y_i$ *by* $Y_i := \eta(S_i) + f(\mathbf{x}_i^{out})$ *for any* $i \geq 0$. *We say that the tuple* $(\mathcal{C}, \eta, f)$ *is* OST-sound *if* $\mathbb{E}[|Y_i|] < \infty$ *for every* $i \geq 0$ *and moreover, at least one of the following conditions (C1)–(C4) holds:*
*(C1) There exists a constant $c$ such that $T \leq c$ with probability 1.*
*(C2) There exists a constant $c$ such that for each $t \in \mathbb{N}$ and run $\rho$ we have* $|Y_{\min\{t, T(\rho)\}}(\rho)| \leq c$ *(i.e., $Y_i(\rho)$ is uniformly bounded along the run).*
*(C3) $\mathbb{E}[T] < \infty$ and there exists a constant $c$ such that for every $t \in \mathbb{N}$ it holds* $\mathbb{E}[|Y_{t+1} - Y_t| \mid \mathcal{F}_t] \leq c$ *(i.e., the expected one-step change of $Y_i$ is uniformly bounded over runtime).*

*(C4) There exist real numbers $M, c_1, c_2, d$ such that (i) for all sufficiently large $n \in \mathbb{N}$ it holds $\mathbb{P}(T > n) \leq c_1 \cdot e^{-c_2 \cdot n}$; and (ii) for all $t \in \mathbb{N}$ it holds $|Y_{n+1} - Y_n| \leq M \cdot n^d$.*

We now present our new proof rule for equivalence refutation in PPs with conditioning. Given a pair $P^1, P^2$ of a.s. terminating PPs that satisfy the bounded cumulative weight property, our proof rule applies the weighted restarting procedure to produce the pCFGs $\mathcal{C}(P^1_{\text{res}})$ and $\mathcal{C}(P^2_{\text{res}})$ and then applies the proof rule of [23] for PPs without conditioning. The following theorem formalizes the instantiation of the proof rule of [23] to the pCFG pair $\mathcal{C}(P^1_{\text{res}})$ and $\mathcal{C}(P^2_{\text{res}})$.

**Theorem 2 (Equivalence refutation for PPs with conditioning).** *Let $P^1, P^2$ be two a.s. terminating programs (possibly with conditioning) satisfying the bounded cumulative weight property. Let the initial states of $\mathcal{C}(P^1_{\text{res}})$ and $\mathcal{C}(P^2_{\text{res}})$ be $(\ell^1_{in}, \mathbf{x}^1_{in})$ and $(\ell^2_{in}, \mathbf{x}^2_{in})$, respectively. Assume that there exist a Borel-measurable function $f \colon \mathbb{R}^{|V_{out}|} \to \mathbb{R}$ and two state functions, $U_f$ for $\mathcal{C}(P^1_{\text{res}})$ and $L_f$ for $\mathcal{C}(P^2_{\text{res}})$ such that the following holds:*
*i) $U_f$ is a UESM for $f$ in $\mathcal{C}(P^1_{\text{res}})$ such that $(\mathcal{C}(P^1_{\text{res}}), U_f, f)$ is OST-sound;*
*ii) $L_f$ is an LESM for $f$ in $\mathcal{C}(P^2_{\text{res}})$ such that $(\mathcal{C}(P^2_{\text{res}}), L_f, f)$ is OST-sound;*
*iii) $U_f(\ell^1_{in}, \mathbf{x}^1_{in}) + f((\mathbf{x}^1_{in})^{out}) < L_f(\ell^2_{in}, \mathbf{x}^2_{in}) + f((\mathbf{x}^2_{in})^{out})$.*
*Then $P^1, P^2$ are not output equivalent,*

The theorem proof follows immediately by the correctness of weighted restarting (Theorem 1) and the correctness of the proof rule in [23, Theorem 5.5]. [4]

While [23] establishes soundness of the above proof rule for PPs *without conditioning*, which here we generalize to PPs *with conditioning*, to the best of our knowledge there is no known proof rule that is also complete. In what follows, we show that the proof rule in Theorem 2 is not only sound, but also complete. This result is new both in the setting of PPs with and without conditioning.

**Theorem 3 (Completeness of the proof rule).** *Let $P^1, P^2$ be a two a.s. terminating programs satisfying the bounded cumulative weight property. Assume that they are not output equivalent. Then there exists a Borel-measurable function $f \colon \mathbb{R}^{|V_{out}|} \to \mathbb{R}$ and two state functions, $U_f$ for $f$ in $\mathcal{C}(P^1_{res})$ and $L_f$ for $f$ in $\mathcal{C}(P^2_{res})$, satisfying the conditions of Theorem 2.*

*Proof.* Since $\mu^{\mathcal{C}(P^1)} \neq \mu^{\mathcal{C}(P^2)}$, by Theorem 1 also $\mu^{\mathcal{C}(P^1_{\text{res}})} \neq \mu^{\mathcal{C}(P^2_{\text{res}})}$. Hence, there is a Borel-measurable set $B \subseteq \mathbb{R}^{V_{out}}$ such that, w.l.o.g., $\mu^{\mathcal{C}(P^1_{\text{res}})}[B] > \mu^{\mathcal{C}(P^2_{\text{res}})}[B]$. Let $p^i_B(\ell, \boldsymbol{x}) = \mathbb{P}^{\mathcal{C}(P^i_{\text{res}})}[Output(B) \mid \text{run initialized in } (\ell, \boldsymbol{x})]$ be the probability of outputting a vector from $B$ if $\mathcal{C}(P^i_{\text{res}})$ starts in $(\ell, \boldsymbol{x})$, for $i \in \{1, 2\}$. We define:
  − A function $f = \mathbb{I}_B$ over outputs;

---

[4] We remark that the program syntax considered in [23] does not include go-to statements. However, the definition of U/LESMs and the proof rule in [23] are formalized with respect to pCFGs. Each go-to statement in a program can simply be modeled as a new edge in the program's pCFG. Since [23] does not make any restriction on the topology of pCFGs, their result remains correct even for pCFGs induced by conditioning-free PPs with go-to statements.

– A state function $U_f(\ell, \boldsymbol{x}) = p_B^1(\ell, \boldsymbol{x}) - f(\boldsymbol{x}^{out})$ in $\mathcal{C}(P_{\mathrm{res}}^1)$;
– A state function $L_f(\ell, \boldsymbol{x}) = p_B^2(\ell, \boldsymbol{x}) - f(\boldsymbol{x}^{out})$ in $\mathcal{C}(P_{\mathrm{res}}^2)$.

We prove that $U_f$ is an UESM $f$ in $\mathcal{C}(P_{\mathrm{res}}^1)$, the argument for $L_f$ is analogous. For any terminal state $(\ell, \boldsymbol{x})$ it holds $p_b^1(\ell, \boldsymbol{x}) = \mathbb{I}_B(\boldsymbol{x}^{out}) = f(\boldsymbol{x}^{out})$, hence $U_f$ is zero on output. Moreover, for any state $(\ell, \boldsymbol{x})$ it holds $U_f(\ell, \boldsymbol{x}) + f(\boldsymbol{x}^{out}) = p_B^1(\ell, \boldsymbol{x})$. Since $p_B^1(\ell, \boldsymbol{x}) = \mathbb{E}[p_B^1(\ell', \mathbf{N})]$ (the standard flow conservation of reachability probabilities), we get that $U_f$ satisfies the expected $f$-decrease condition.

Regarding OST soundness, note that $U_f(\ell, \boldsymbol{x}) + f(\boldsymbol{x}^{out}) = p_B^1 \in [0,1]$ for any state $(\ell, \boldsymbol{x})$. Hence, $(\mathcal{C}(P_{\mathrm{res}}^1), U_f, f)$ is OST-sound – it satisfies condition (C2) – and similarly for $(\mathcal{C}(P_{\mathrm{res}}^2), L_f, f)$.

It remains to verify condition iii) from Theorem 2. We have

$$U_f(\ell_{in}^1, \mathbf{x}_{in}^1) + f((\mathbf{x}_{in}^1)^{out}) = p_B^1(\ell_{in}, \mathbf{x}_{in}) = \mu^{\mathcal{C}(P_{\mathrm{res}}^1)}[B] < \mu^{\mathcal{C}(P_{\mathrm{res}}^2)}[B]$$
$$= p_B^2(\ell_{in}, \mathbf{x}_{in}) = L_f(\ell_{in}^1, \mathbf{x}_{in}^1) + f((\mathbf{x}_{in}^1)^{out}).$$

*Example 4.* To conclude this section, we illustrate our proof rule on our running example in Fig. 1. Consider the PPs $(P_{\mathrm{res}}^1, P_{\mathrm{res}}^2)$ obtained by weighted restarting in Fig. 2, and the function $f$ over outputs, the UESM $U_f$ for $f$ in $\mathcal{C}(P_{\mathrm{res}}^1)$, and the LESM $L_f$ for $f$ in $\mathcal{C}(P_{\mathrm{res}}^2)$ constructed in Example 3. The triples $(\mathcal{C}(P_{\mathrm{res}}^1), U_f, f)$ and $(\mathcal{C}(P_{\mathrm{res}}^2), L_f, f)$ are both OST-sound and satisfy (C2) in Definition 2, as all variable values in $P_{\mathrm{res}}^1$ and $P_{\mathrm{res}}^2$ are bounded. Moreover, we have $U_f(\ell_{in}^1, \mathbf{x}_{in}^1) + f((\mathbf{x}_{in}^1)^{out}) = 500 < 600 = L_f(\ell_{in}^2, \mathbf{x}_{in}^2) + f((\mathbf{x}_{in}^2)^{out})$. Hence, $(f, U_f, L_f)$ satisfy all conditions of Theorem 2, and PPs in Fig. 1 are not output equivalent.

# 5   Automated Equivalence Refutation

We now present our algorithm for equivalence refutation in PPs that may contain conditioning instructions. Given a PP pair $(P^1, P^2)$, the algorithm first applies weighted restarting to translate them into output equivalent conditioning-free PPs $(P_{\mathrm{res}}^1, P_{\mathrm{res}}^2)$. Then, the algorithm applies the equivalence refutation procedure of [23] for PPs without conditioning. Hence, given that our algorithm builds on that of [23], we only present an overview of the algorithm and refer the reader to [23] for details. However, this section presents two important novel results:

1. *Semi-completeness guarantees.* Building on our result in Section 4 which shows that our proof rule is sound and complete, we show that our algorithm for equivalence refutation is sound and *semi-complete*. In particular, the algorithm is guaranteed to refute equivalence of PPs whenever non-equivalence can be witnessed by a function $f$ and a pair of U/LESMs that can be specified via polynomial functions of some maximal polynomial degree $D$, and that satisfy OST-soundness condition (C2) in Definition 2.
2. *Streamlined algorithm for bounded termination time PPs.* We show that the algorithm can be streamlined for the class of PPs whose (1) termination time is bounded by some constant value $T > 0$, and (2) all sampling instructions consider probability distributions of bounded support. The first assumption is common and satisfied by most PPs considered in statistical inference literature, including all our benchmarks in Section 6.

*Algorithm assumptions.* In order to allow for a fully automated equivalence refu-
tation, we need to impose several assumptions on our PPs:

- *Polynomial arithmetic.* We require all arithmetic expressions appearing in
  PPs to be polynomials over program variables. This restriction is necessary
  for the full automation of our method. We also require that each proba-
  bility distribution appearing in sampling instructions in pCFGs has *finite
  moments*, i.e. for each $p \in \mathbb{N}$, the $p$-th moment $m_\delta(p) = \mathbb{E}_{X \sim \delta}[|X|^p]$ is finite
  and can be computed by the algorithm. This is a standard assumption in
  static PP analysis and allows for most standard probability distributions.
- *Almost-sure termination.* For the equivalence refutation problem to be well-
  defined and for our proof rule to be applicable, we require both PPs to
  be a.s. terminating. In polynomial arithmetic PPs, this requirement can be
  *automatically checked* by using the method of [17].
- *Bounded cumulative weight property.* To automatically check this property,
  we can use an off-the-shelf invariant generator, e.g. [31,56,18], to generate
  an invariant which proves that $W \leq M$ holds for the weight variable $W$ and
  some sufficiently large constant value $M$.
- *Supporting linear invariants.* Recall that we defined UESMs and LESMs
  with respect to supporting invariants. To that end, before proceeding to
  equivalence refutation, our algorithm first synthesizes *linear invariants* $I_1$
  and $I_2$. This can be fully automated by using existing linear invariant gen-
  erators [31,56], as done in our implementation. For the purpose of invariant
  generation, sampling instructions are semantically over-approximated via
  non-determinism, hence ensuring that the computed invariants are sound.

*Template-based synthesis algorithm.* Our algorithm first uses weighted restarting
to produce $P_{\text{res}}^1$ and $P_{\text{res}}^2$, and constructs their pCFGs $\mathcal{C}(P_{\text{res}}^1)$ and $\mathcal{C}(P_{\text{res}}^2)$. It then
follows a template-based synthesis approach and simultaneously synthesizes the
triple $(f, U_f, L_f)$ required by our proof rule in Theorem 2. The algorithm pro-
ceeds in four steps. In Step 1, it fixes templates for $f$, $U_f$ and $L_f$, where the
templates are symbolic polynomials over pCFG variables of some maximal poly-
nomial degree $D$, an algorithm parameter provided by the user. In Step 2, the
algorithm collects a system of constraints over the symbolic polynomial template
variables, which together encode that $(f, U_f, L_f)$ satisfy all the requirements and
give rise to a correct instance of the proof rule in Theorem 2. At this point, the
resulting constraints contain polynomial expressions and quantifier alternation,
which makes their solving challenging. Thus, in Step 3, the algorithm trans-
lates the collected system of constraints into a purely *existentially quantified*
system of *linear* constraints over the symbolic template variables. This is done
via Handelman's theorem [41], analogously to previous template-based synthesis
PP analyses [63,4,67]. In Step 4, the algorithm solves the system of linear con-
straints via a linear programming (LP) tool. If the system is feasible, its solution
yields a concrete instance of $(f, U_f, L_f)$ from Theorem 2 and the algorithm re-
ports "Not output-equivalent". Due to reducing the synthesis to an LP instance,
the runtime complexity of our algorithm is polynomial in the size of the pCFGs,
for any fixed value of the polynomial degree parameter $D$. Finally, the proof of

Theorem 3 establishes completeness of our proof rule with (C2) OST-soundness condition, and here we show semi-completeness of our algorithm.

**Theorem 4 (Algorithm correctness, complexity and semi-completeness, proof in [24]).** *The algorithm is correct. If the algorithm outputs "Not output-equivalent", then $(P^1, P^2)$ are indeed not output-equivalent and the computed triple $(f, U_f, L_f)$ forms a correct instance of the proof rule in Theorem 2.*

*Furthermore, the runtime complexity of the algorithm is polynomial in the size of the programs' pCFGs, for any fixed value of the polynomial degree $D$.*

*Finally, the algorithm is semi-complete, meaning that if there exists a valid triple $(f, U_f, L_f)$ that can be specified via polynomial expressions of degree at most $D$ and that satisfy OST-soundness condition (C2) in Definition 2, then the algorithm is guaranteed to output "Not output-equivalent".*

*Streamlined algorithm for bounded termination time PPs.* We conclude this section by showing that the algorithm can be streamlined for the class of PPs whose (1) termination time is bounded by some $T > 0$, and (2) all sampling instructions consider probability distributions of bounded support. We show that for this class of PPs, any polynomial function $f$ over outputs and any polynomial UESM $U_f$ (resp. LESM $L_f$) for $f$ are guaranteed to be (C2) OST-sound. Hence, the algorithm does not need to check OST-soundness, which significantly simplifies the system of constraints that needs to be solved by the SMT-solver.

**Theorem 5 (Proof in [24]).** *Let $P$ be a PP with (1) termination time bounded by some $T > 0$, and (2) all sampling instructions considering probability distributions of bounded support. Then, for any polynomial function $f$ over outputs, polynomial UESM $U_f$ for $f$ in $\mathcal{C}(P_{res}^1)$ and LESM $L_f$ for $f$ in $\mathcal{C}(P_{res}^2)$, $(\mathcal{C}(P_{res}^1), U_f, f)$ and $(\mathcal{C}(P_{res}^2), L_f, f)$ satisfy the (C2) OST-soundness condition.*

## 6    Experimental Results

*Implementation.* We implemented a prototype of our equivalence refutation algorithm. The prototype tool is implemented in Java. Gurobi [40] is used for solving the LP instances while STING [56] and ASPIC [31] are used for generating supporting invariants. For each input program pair, the tool tries to perform polynomial template-based synthesis by using polynomial degrees 1 to 5 in order. All experiments were conducted on an Ubuntu 24.04 machine with an 11th Gen Intel Core i5 CPU and 16 GB RAM with a timeout of 10 minutes.

*Baseline.* To the best of our knowledge, no prior work has considered formal and automated equivalence refutation for PPs with conditioning. However, a viable alternative approach to refuting PP equivalence would be to first compute the output distributions of the two PPs via exact symbolic inference, and then to use mathematical computation software to show that these distributions are different. Hence, we compare our method against such a baseline. We use the state-of-the-art exact inference tool PSI [34] for computing output distributions, and Mathematica [66] for proving that they are different.

Table 1: Experimental results showing the performance of our tool and the baseline for refuting equivalence of PPs. A ✓ in the "Eq. Ref." column represents that the tool successfully refuted equivalence of the two input programs, whereas "TO" and "NA" stand for "Timeout" and "Not Applicable", respectively. The `Distance` column shows the Kantorovich distance lower bound computed by out tool for the similarity refutation problem (see Section 1).

| | Name | Our Method | | | | PSI + Mathematica | |
|---|---|---|---|---|---|---|---|
| | | Eq. Ref. | Time(s) | Distance | Time(s) | Eq. Ref. | Time(s) |
| Benchmarks from [34] | bertrand | ✓ | 0.22 | 0.42 | 0.30 | ✓ | 0.86 |
| | burglarAlarm | ✓ | 2.01 | 0.05 | 1.39 | ✓ | 1.04 |
| | coinBiasSmall | ✓ | 36.92 | TO | - | ✓ | 1.07 |
| | coinPattern | ✓ | 4.50 | 0.04 | 56.33 | ✓ | 0.86 |
| | coins | ✓ | 0.46 | 0.12 | 0.95 | ✓ | 0.83 |
| | ev-model1 | ✓ | 0.45 | 0.16 | 0.81 | ✓ | 0.85 |
| | ev-model2 | ✓ | 0.75 | 0.20 | 0.92 | ✓ | 0.80 |
| | gossip | ✓ | 0.62 | 0.1 | 0.63 | ✓ | 1.01 |
| | grass | ✓ | 1.46 | 0.06 | 8.15 | ✓ | 1.02 |
| | murderMystery | ✓ | 0.53 | 0.004 | 1.24 | ✓ | 0.84 |
| | noisyOr | ✓ | 1.17 | 0.17 | 2.15 | ✓ | 1.98 |
| | twoCoins | ✓ | 0.66 | 0.12 | 1.86 | ✓ | 0.93 |
| Benchmarks from [23] | Simple Example | ✓ | 7.28 | 381.71 | 12.15 | TO | - |
| | Nested Loop | TO | - | TO | - | TO | - |
| | Random Walk | ✓ | 32.01 | TO | - | TO | - |
| | Goods Discount | ✓ | 0.76 | 0.08 | 1.05 | TO | - |
| | Pollutant Disposal | ✓ | 3.08 | 327.27 | 3.43 | TO | - |
| | 2D Robot | ✓ | 0.77 | TO | - | TO | - |
| | Bitcoin Mining | ✓ | 7.28 | TO | - | TO | - |
| | Bitcoin Mining Pool | ✓ | 13.248 | TO | - | TO | - |
| | Species Fight | TO | - | TO | - | TO | - |
| | Queuing Network | TO | - | TO | - | TO | - |
| | Coupon1 | ✓ | 3.86 | 0.47 | 4.64 | TO | - |
| | Coupon4 | ✓ | 169.47 | TO | - | TO | - |
| | random_walk_1d_intvalued | ✓ | 6.24 | TO | - | TO | - |
| | random_walk_1d_realvalued | TO | - | TO | - | TO | - |
| | random_walk_1d_adversary | TO | - | TO | - | TO | - |
| | random_walk_2d_demonic | ✓ | 25.99 | TO | - | TO | - |
| | random_walk_2d_variant | TO | - | TO | - | TO | - |
| | retransmission | ✓ | 2.43 | TO | - | TO | - |
| Benchmarks from [64] | add-uni | ✓ | 10.25 | 0.01 | 17.48 | TO | - |
| | cav-ex-5 | TO | - | TO | - | TO | - |
| | cav-ex-7 | ✓ | 0.19 | 10.00 | 0.20 | TO | - |
| | pdmb-v3 | ✓ | 60.27 | 2100.00 | 60.27 | NA | - |
| | race | ✓ | 1.40 | 2350.42 | 1.39 | NA | - |
| | rdwalk-v12 | ✓ | 0.85 | 100.00 | 0.75 | NA | - |
| | rdwalk-v13 | ✓ | 1.36 | 0.01 | 16.53 | NA | - |
| | rdwalk-v14 | ✓ | 0.77 | 100.00 | 0.80 | NA | - |
| | rdwalk-v23 | ✓ | 0.78 | 100.00 | 0.90 | NA | - |
| | rdwalk-v24 | ✓ | 1.06 | 200.00 | 0.90 | NA | - |
| | rdwalk-v34 | ✓ | 2.19 | 100.00 | 1.09 | NA | - |
| | Success Count | 34 | - | 26 | - | 12 | - |
| | Average Runtime | - | 11.8 | - | 7.63 | - | 1.01 |

*Benchmarks.* While our method is applicable to PPs with unbounded loops, PSI (and hence our baseline) is restricted to PPs with statically bounded loops. Hence, for all our benchmarks, we bound the maximal number of iterations of each loop by 1000. Note that, while this modification ensures that all loops admit an upper bound on the number of iterations, it still allows loops to terminate early (i.e. in strictly less than 1000 iterations) if the termination condition is met early. Hence, executions of programs are not aligned by this modification, making them challenging in structure for relational analysis.

We ran our tool and the baseline on three sets of benchmarks. For each benchmark in the first two sets, we obtain a pair of non-output equivalent PPs by slightly perturbing exactly one of the sampling instructions appearing in the original PP. Benchmarks in the third set already come in pairs:

1. We consider 12 PPs with discrete probability distributions from the PSI benchmark suite (taken from the repository of [11]). We selected benchmarks supported by our syntax, e.g., avoiding reading data from external .CSV files.
2. We consider 18 loopy programs from [23] on equivalence refutation in PPs without conditioning (benchmarks originating from [63,44]). In order to introduce conditioning statements, to each program we add an `observe` statement at the end, which observes a bound on one of the output variables.
3. Lastly, we consider the benchmarks used in [64] for symbolic inference in PPs with conditioning. Specifically, we consider pairs of benchmarks that already have several versions in the repository of [64] (for example there are four versions of the `rdwalk` program that were different only in one parameter; we ran our program on all 6 possible program pairs). All the programs in this benchmark set contain `score` statements. Note that PSI (and hence the baseline) does not support `score` statements. However, some benchmarks from this repository contain `"if (g) score(1) else score(0) fi"` fragments, which we replace by `observe(g)` in order to make PSI applicable.

*Results.* Our experimental results are shown in Table 1. We observe the following:

– On the first benchmark set in which PPs terminate in a small number of steps, both our method and the baseline manage to refute equivalence of all 12 PP pairs. Our method is faster in 8/12 cases, whereas the baseline is faster in 4/12 cases. However, in some of these cases our method is significantly slower (e.g. `coinBiasSmall`), because the synthesis algorithm needed to use degree 5 polynomial templates to refute equivalence. However, our method overall shows a better performance than the baseline. Moreover, our method successfully computes Kantorovich distance lower bounds in 11/12 cases.
– On the second benchmark set in which PPs take much longer time to terminate (e.g loops may be executed for up to 1000 iterations), our method refutes equivalence in 12/18 cases, whereas the baseline fails in all cases. This shows an advantage of our method. The baseline is based on exact symbolic inference, meaning that it needs to symbolically execute the program which is a computational bottleneck when the program runtime is large. In contrast, our method is a static analysis method and does not suffer from this limitation. When it comes to computing lower bounds on Kantorovich distance,

our tool succeeds in only 4/18 cases. We believe this is due to (i) the invariant generation tool StIng failing to compute invariants in several cases, or (ii) the UESM/LESMs needed being non-polynomial (e.g. for `Species Fight`).

– On the third benchmark set, our tool successfully refutes equivalence and computes lower bounds on distance in 10/11 cases. These results demonstrate the applicability of our method in the presence of both `observe` and `score` statements. The baseline solves no cases – fails on 3 whereas it is not applicable in 8 cases as PSI does not support `score` statements.

*Limitations.* In our experiments, we also observed a limitation of our algorithm, addressing which is an interesting direction of future work. Supporting invariants play an important role for guiding the U/LESM synthesis algorithm. Without good supporting invariants, the synthesis algorithm might fail. Generating good invariants can sometimes be time consuming. For example, our tool takes 169 seconds to solve the `coupon4` benchmark in our experiments, because STING and ASPIC invariant generators take more than 2 minutes to generate an invariant.

## 7    Conclusion

We presented a new method for refuting equivalence of PPs with conditioning. To the best of our knowledge, we presented the first method that is fully automated, provides formal guarantees, and is sound in the presence of conditioning instructions. Moreover, we show our proof rule is also complete and our algorithm is semi-complete. Our prototype implementation demonstrates the applicability of our method to a range of pairs of PPs with conditioning. An interesting direction of future work would be to consider the automation of our method for non-polynomial programs. Another interesting direction would be to consider the applicability of the weighted restarting procedure to other static analyses in PPs with conditioning, towards reducing them to the conditioning-free setting.

## Acknowledgments

## References

1. Agrawal, S., Chatterjee, K., Novotný, P.: Lexicographic ranking supermartingales: an efficient approach to termination of probabilistic programs. Proc. ACM Program. Lang. **2**(POPL) (2018)
2. Aguirre, A., Barthe, G., Hsu, J., Kaminski, B.L., Katoen, J., Matheja, C.: A pre-expectation calculus for probabilistic sensitivity. Proc. ACM Program. Lang. **5**(POPL), 1–28 (2021). https://doi.org/10.1145/3434333

3. Albarghouthi, A., Hsu, J.: Synthesizing coupling proofs of differential privacy. Proc. ACM Program. Lang. **2**(POPL), 58:1–58:30 (2018). https://doi.org/10.1145/3158146

4. Asadi, A., Chatterjee, K., Fu, H., Goharshady, A.K., Mahdavi, M.: Polynomial reachability witnesses via stellensätze. In: Freund, S.N., Yahav, E. (eds.) PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021. pp. 772–787. ACM (2021). https://doi.org/10.1145/3453483.3454076

5. Avanzini, M., Moser, G., Schaper, M.: A modular cost analysis for probabilistic programs. Proc. ACM Program. Lang. **4**(OOPSLA) (nov 2020). https://doi.org/10.1145/3428240

6. Barthe, G., Espitau, T., Grégoire, B., Hsu, J., Strub, P.: Proving expected sensitivity of probabilistic programs. Proc. ACM Program. Lang. **2**(POPL), 57:1–57:29 (2018). https://doi.org/10.1145/3158145

7. Barthe, G., Gaboardi, M., Grégoire, B., Hsu, J., Strub, P.: Proving differential privacy via probabilistic couplings. In: Grohe, M., Koskinen, E., Shankar, N. (eds.) Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016. pp. 749–758. ACM (2016). https://doi.org/10.1145/2933575.2934554

8. Barthe, G., Gaboardi, M., Hsu, J., Pierce, B.C.: Programming language techniques for differential privacy. ACM SIGLOG News **3**(1), 34–53 (2016). https://doi.org/10.1145/2893582.2893591

9. Batu, T., Fortnow, L., Rubinfeld, R., Smith, W.D., White, P.: Testing closeness of discrete distributions. J. ACM **60**(1), 4:1–4:25 (2013). https://doi.org/10.1145/2432622.2432626

10. Batz, K., Chen, M., Junges, S., Kaminski, B.L., Katoen, J., Matheja, C.: Probabilistic program verification via inductive synthesis of inductive invariants. In: TACAS (2023)

11. Beutner, R., Ong, C.L., Zaiser, F.: Guaranteed bounds for posterior inference in universal probabilistic programming. In: PLDI. pp. 536–551. ACM (2022)

12. Bichsel, B., Gehr, T., Vechev, M.T.: Fine-grained semantics for probabilistic programs. In: Ahmed, A. (ed.) Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10801, pp. 145–185. Springer (2018). https://doi.org/10.1007/978-3-319-89884-1_6, https://doi.org/10.1007/978-3-319-89884-1_6

13. Canonne, C.L.: A survey on distribution testing: Your data is big. but is it blue? Theory of Computing pp. 1–100 (2020)

14. Chakarov, A., Sankaranarayanan, S.: Probabilistic program analysis with martingales. In: CAV (2013)

15. Chakraborty, S., Meel, K.S.: On testing of uniform samplers. In: The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019. pp. 7777–7784. AAAI Press (2019). https://doi.org/10.1609/aaai.v33i01.33017777

16. Chan, S., Diakonikolas, I., Valiant, P., Valiant, G.: Optimal algorithms for testing closeness of discrete distributions. In: Chekuri, C. (ed.) Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA

2014, Portland, Oregon, USA, January 5-7, 2014. pp. 1193–1203. SIAM (2014). https://doi.org/10.1137/1.9781611973402.88

17. Chatterjee, K., Fu, H., Goharshady, A.K.: Termination analysis of probabilistic programs through positivstellensatz's. In: CAV (2016)

18. Chatterjee, K., Fu, H., Goharshady, A.K., Goharshady, E.K.: Polynomial invariant generation for non-deterministic recursive programs. In: PLDI. pp. 672–687. ACM (2020)

19. Chatterjee, K., Fu, H., Novotný, P., Hasheminezhad, R.: Algorithmic analysis of qualitative and quantitative termination problems for affine probabilistic programs. ACM Trans. Program. Lang. Syst. **40**(2) (2018)

20. Chatterjee, K., Goharshady, A.K., Meggendorfer, T., Zikelic, D.: Sound and complete certificates for quantitative termination analysis of probabilistic programs. In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I. Lecture Notes in Computer Science, vol. 13371, pp. 55–78. Springer (2022). https://doi.org/10.1007/978-3-031-13185-1_4

21. Chatterjee, K., Goharshady, A.K., Meggendorfer, T., Zikelic, D.: Quantitative bounds on resource usage of probabilistic programs. Proc. ACM Program. Lang. **8**(OOPSLA1), 362–391 (2024). https://doi.org/10.1145/3649824, https://doi.org/10.1145/3649824

22. Chatterjee, K., Goharshady, E.K., Novotný, P., Zárevúcky, J., Zikelic, D.: On lexicographic proof rules for probabilistic termination. In: Huisman, M., Pasareanu, C.S., Zhan, N. (eds.) Formal Methods - 24th International Symposium, FM 2021, Virtual Event, November 20-26, 2021, Proceedings. Lecture Notes in Computer Science, vol. 13047, pp. 619–639. Springer (2021). https://doi.org/10.1007/978-3-030-90870-6_33

23. Chatterjee, K., Goharshady, E.K., Novotný, P., Zikelic, D.: Equivalence and similarity refutation for probabilistic programs. Proc. ACM Program. Lang. **8**(PLDI), 2098–2122 (2024). https://doi.org/10.1145/3656462, https://doi.org/10.1145/3656462

24. Chatterjee, K., Goharshady, E.K., Novotný, P., Đorđe Žikelić: Refuting equivalence in probabilistic programs with conditioning (2025), https://arxiv.org/abs/2501.06579

25. Chatterjee, K., Novotný, P., Zikelic, D.: Stochastic invariants for probabilistic termination. In: Castagna, G., Gordon, A.D. (eds.) Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017. pp. 145–160. ACM (2017). https://doi.org/10.1145/3009837.3009873

26. Chen, J., He, F.: Proving almost-sure termination by omega-regular decomposition. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'20). p. 869–882. PLDI 2020, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3385412.3386002

27. Chen, M., Katoen, J., Klinkenberg, L., Winkler, T.: Does a program yield the right distribution? - verifying probabilistic programs via generating functions. In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I. Lecture Notes in Computer Science, vol. 13371, pp. 79–101. Springer (2022). https://doi.org/10.1007/978-3-031-13185-1_5

28. Cusumano-Towner, M.F., Mansinghka, V.K.: AIDE: an algorithm for measuring the accuracy of probabilistic inference algorithms. In: NIPS. pp. 3000–3010 (2017)

29. Das, A., Wang, D., Hoffmann, J.: Probabilistic resource-aware session types. Proc. ACM Program. Lang. **7**(POPL) (jan 2023). https://doi.org/10.1145/3571259

30. Domke, J.: An easy to interpret diagnostic for approximate inference: Symmetric divergence over simulations. CoRR **abs/2103.01030** (2021)

31. Feautrier, P., Gonnord, L.: Accelerated invariant generation for C programs with aspic and c2fsm. In: TAPAS@SAS (2010)

32. Fioriti, L.M.F., Hermanns, H.: Probabilistic Termination: Soundness, Completeness, and Compositionality. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2015). pp. 489–501 (2015). https://doi.org/10.1145/2676726.2677001

33. Foster, N., Kozen, D., Mamouras, K., Reitblatt, M., Silva, A.: Probabilistic netkat. In: Thiemann, P. (ed.) Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9632, pp. 282–309. Springer (2016). https://doi.org/10.1007/978-3-662-49498-1_12

34. Gehr, T., Misailovic, S., Vechev, M.T.: PSI: exact symbolic inference for probabilistic programs. In: CAV (2016)

35. Ghahramani, Z.: Probabilistic machine learning and artificial intelligence. Nat. **521**(7553), 452–459 (2015). https://doi.org/10.1038/nature14541

36. Goldblatt, R., Jackson, M.: Well-structured program equivalence is highly undecidable. ACM Trans. Comput. Log. **13**(3), 26:1–26:8 (2012). https://doi.org/10.1145/2287718.2287726, https://doi.org/10.1145/2287718.2287726

37. Gregersen, S.O., Aguirre, A., Haselwarter, P.G., Tassarotti, J., Birkedal, L.: Asynchronous probabilistic couplings in higher-order separation logic. Proc. ACM Program. Lang. **8**(POPL), 753–784 (2024). https://doi.org/10.1145/3632868

38. Grosse, R.B., Ancha, S., Roy, D.M.: Measuring the reliability of MCMC inference with bidirectional monte carlo. In: NIPS. pp. 2451–2459 (2016)

39. Grosse, R.B., Ghahramani, Z., Adams, R.P.: Sandwiching the marginal likelihood using bidirectional monte carlo. CoRR **abs/1511.02543** (2015)

40. Gurobi Optimization, LLC: Gurobi Optimizer Reference Manual (2023), https://www.gurobi.com

41. Handelman, D.: Representing polynomials by positive linear functions on compact convex polyhedra. Pacific Journal of Mathematics **132**(1) (1988)

42. Huang, Z., Wang, Z., Misailovic, S.: Psense: Automatic sensitivity analysis for probabilistic programs. In: Lahiri, S.K., Wang, C. (eds.) Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings. Lecture Notes in Computer Science, vol. 11138, pp. 387–403. Springer (2018). https://doi.org/10.1007/978-3-030-01090-4_23

43. Katoen, J.P., Gretz, F., Jansen, N., Kaminski, B.L., Olmedo, F.: Understanding Probabilistic Programs, pp. 15–32. Springer International Publishing, Cham (2015). https://doi.org/10.1007/978-3-319-23506-6_4

44. Kura, S., Urabe, N., Hasuo, I.: Tail probabilities for randomized program runtimes via martingales for higher moments. In: TACAS (2). Lecture Notes in Computer Science, vol. 11428, pp. 135–153. Springer (2019)

45. Legay, A., Murawski, A.S., Ouaknine, J., Worrell, J.: On automated verification of probabilistic programs. In: TACAS. Lecture Notes in Computer Science, vol. 4963, pp. 173–187. Springer (2008)

46. Majumdar, R., Sathiyanarayana, V.R.: Positive almost-sure termination: Complexity and proof rules. Proc. ACM Program. Lang. **8**(POPL) (jan 2024). https://doi.org/10.1145/3632879

47. McIver, A., Morgan, C.: Abstraction, Refinement and Proof for Probabilistic Systems. Monographs in Computer Science, Springer (2005)

48. McIver, A., Morgan, C.: Correctness by construction for probabilistic programs. In: ISoLA (1). Lecture Notes in Computer Science, vol. 12476, pp. 216–239. Springer (2020)

49. McIver, A., Morgan, C., Kaminski, B.L., Katoen, J.: A new proof rule for almost-sure termination. Proc. ACM Program. Lang. **2**(POPL), 33:1–33:28 (2018). https://doi.org/10.1145/3158121

50. Meyn, S.P., Tweedie, R.L.: Markov chains and stochastic stability. Springer Science & Business Media (2012)

51. Murawski, A.S., Ouaknine, J.: On probabilistic program equivalence and refinement. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005 - Concurrency Theory, 16th International Conference, CONCUR 2005, San Francisco, CA, USA, August 23-26, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3653, pp. 156–170. Springer (2005). https://doi.org/10.1007/11539452_15, https://doi.org/10.1007/11539452_15

52. Ngo, V.C., Carbonneaux, Q., Hoffmann, J.: Bounded expectations: resource analysis for probabilistic programs. In: Foster, J.S., Grossman, D. (eds.) Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018. pp. 496–512. ACM (2018). https://doi.org/10.1145/3192366.3192394

53. Olmedo, F., Gretz, F., Jansen, N., Kaminski, B.L., Katoen, J., McIver, A.: Conditioning in probabilistic programming. ACM Trans. Program. Lang. Syst. **40**(1), 4:1–4:50 (2018). https://doi.org/10.1145/3156018

54. Olmedo, F., Gretz, F., Jansen, N., Kaminski, B.L., Katoen, J., McIver, A.: Conditioning in probabilistic programming. ACM Trans. Program. Lang. Syst. **40**(1), 4:1–4:50 (2018). https://doi.org/10.1145/3156018

55. Sankaranarayanan, S., Chakarov, A., Gulwani, S.: Static analysis for probabilistic programs: inferring whole program properties from finitely many paths. In: Boehm, H., Flanagan, C. (eds.) ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013. pp. 447–458. ACM (2013). https://doi.org/10.1145/2491956.2462179

56. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Constraint-based linear-relations analysis. In: SAS (2004)

57. Takisaka, T., Oyabu, Y., Urabe, N., Hasuo, I.: Ranking and repulsing supermartingales for reachability in randomized programs. ACM Trans. Program. Lang. Syst. **43**(2), 5:1–5:46 (2021). https://doi.org/10.1145/3450967

58. Takisaka, T., Zhang, L., Wang, C., Liu, J.: Fairness in ranking supermartingales. CoRR **abs**/**2304.11363** (2023). https://doi.org/10.48550/ARXIV.2304.11363

59. Thrun, S.: Probabilistic algorithms in robotics. AI Mag. **21**(4), 93–109 (2000). https://doi.org/10.1609/aimag.v21i4.1534

60. Villani, C.: Topics in optimal transportation, vol. 58. American Mathematical Soc. (2021)

61. Wang, D., Hoffmann, J., Reps, T.W.: Central moment analysis for cost accumulators in probabilistic programs. In: Freund, S.N., Yahav, E. (eds.) PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021. pp. 559–573. ACM (2021). https://doi.org/10.1145/3453483.3454062

62. Wang, P., Fu, H., Chatterjee, K., Deng, Y., Xu, M.: Proving expected sensitivity of probabilistic programs with randomized variable-dependent termination time. Proc. ACM Program. Lang. **4**(POPL), 25:1–25:30 (2020). https://doi.org/10.1145/3371093

63. Wang, P., Fu, H., Goharshady, A.K., Chatterjee, K., Qin, X., Shi, W.: Cost analysis of nondeterministic probabilistic programs. In: PLDI (2019)

64. Wang, P., Yang, T., Fu, H., Li, G., Ong, C.L.: Static posterior inference of bayesian probabilistic programming via polynomial solving. Proc. ACM Program. Lang. **8**(PLDI), 1361–1386 (2024). https://doi.org/10.1145/3656432, https://doi.org/10.1145/3656432

65. Williams, D.: Probability with Martingales. Cambridge mathematical textbooks, Cambridge University Press (1991)

66. Wolfram Research, Inc.: Mathematica 13.2 (2022), https://www.wolfram.com

67. Zikelic, D., Chang, B.E., Bolignano, P., Raimondi, F.: Differential cost analysis with simultaneous potentials and anti-potentials. In: Jhala, R., Dillig, I. (eds.) PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022. pp. 442–457. ACM (2022). https://doi.org/10.1145/3519939.3523435