

This is the Moment for Probabilistic Loops

MARCEL MOOSBRUGGER, TU Wien, Austria

MIROSLAV STANKOVIČ, TU Wien, Austria

EZIO BARTOCCI, TU Wien, Austria

LAURA KOVÁCS, TU Wien, Austria

We present a novel static analysis technique to derive higher moments for program variables for a large class of probabilistic loops with potentially uncountable state spaces. Our approach is fully automatic, meaning it does not rely on externally provided invariants or templates. We employ algebraic techniques based on linear recurrences and introduce program transformations to simplify probabilistic programs while preserving their statistical properties. We develop power reduction techniques to further simplify the polynomial arithmetic of probabilistic programs and define the theory of moment-computable probabilistic loops for which higher moments can precisely be computed. Our work has applications towards recovering probability distributions of random variables and computing tail probabilities. The empirical evaluation of our results demonstrates the applicability of our work on many challenging examples.

CCS Concepts: • **Mathematics of computing** → **Markov processes**; • **Computing methodologies** → **Symbolic and algebraic algorithms**.

1 INTRODUCTION

Probabilistic programming languages enrich classical imperative or functional languages with native primitives to draw samples from random distributions, such as Bernoulli, Uniform, and Normal distributions. The resulting probabilistic programs (PPs) [Barthe et al. 2020; Kozen 1985] embed uncertain quantities, represented by random variables, within standard program control flows. As such, PPs offer a unifying framework to naturally encode probabilistic machine learning models [Ghahramani 2015], for example Bayesian networks [Kaminski et al. 2016], into programs. Moreover, PPs enable programmers to handle uncertainty resulting from sensor measurements and environmental perturbations in cyber-physical systems [Chou et al. 2020; Selyunin et al. 2015]. Other notable examples of PPs include the implementation of cryptographic [Barthe et al. 2012a] and privacy [Barthe et al. 2012b] protocols, as well as randomized algorithms [Motwani and Raghavan 1995] such as Herman’s self-stabilization protocol [Herman 1990] for recovering from faults in a process token ring – see our example in Figure 2.

Analysis of PPs. The random nature of PPs makes their functional analysis very challenging as one needs to reason about probability distributions of random variables instead of computing with their exact values [Barthe et al. 2020]. A standard approach towards handling probability distributions associated with random variables is to estimate such distributions by sampling PPs using Monte Carlo simulation techniques [Hastings 1970]. While such approaches work well for statistical model checking [Younes and Simmons 2006], they are not suitable for the analysis of PPs with potentially infinite program loops as simulating infinite-state behavior is not viable. Moreover, even for PPs with finitely many states, simulation-based analysis becomes computationally expensive when a PPs branch is rarely executed.

With the aim of precisely, and not just approximately, handling random variables, probabilistic model checking [Dehnert et al. 2017; Kwiatkowska et al. 2011] became a prominent approach in

Authors’ addresses: Marcel Moosbrugger, TU Wien, Vienna, Austria, marcel.moosbrugger@tuwien.ac.at; Miroslav Stanković, TU Wien, Vienna, Austria, miroslav.stankovic@tuwien.ac.at; Ezio Bartocci, TU Wien, Vienna, Austria, ezio.bartocci@tuwien.ac.at; Laura Kovács, TU Wien, Vienna, Austria, laura.kovacs@tuwien.ac.at.

the analysis of PPs with finite state spaces. For analyzing unbounded PPs, these techniques would however require non-trivial user guidance, in terms of assertion templates and/or invariants.

In this paper, we address the challenge of precisely analyzing, and even recovering, probability distributions induced by PPs with both countably and uncountably infinite state spaces. We do so by extending both expressivity and automation of the state-of-the-art in PP analysis: We (i) focus on PPs with probabilistic infinite loops (see Figure 1) and (ii) fully automate the analysis of such loops by computing exact higher-order statistical moments of program variables x parameterized by a loop counter n .

Functional representations $f(n)$ for a program variable x , with $f(n)$ characterizing the k th moment $\mathbb{E}(x_n^k)$ of x at iteration n , can be interpreted as a quantitative invariant $\mathbb{E}(x_n^k) = f(n)$. Inferring quantitative invariants is arguably not novel. On the contrary, it is one of the most challenging aspects of PP analysis, dating back to the seminal works of [Katoen et al. 2010; McIver and Morgan 2005] introducing the weakest pre-expectations calculus. Template-based approaches to discover invariants or (super-)martingales emerged [Barthe et al. 2016; Kura et al. 2019] by translating the invariant generation problem into a constraint solving one. The derived quantitative invariants are generally provided in terms of the expected values [Chakarov and Sankaranarayanan 2014; Katoen et al. 2010; McIver and Morgan 2005]. Nevertheless, the expected value alone — also referred to as the *first moment* — provides only partial information about the underlying probability distribution. This motivates the critical importance of higher moments for PP analysis [Bartocci et al. 2020b; Kura et al. 2019; Stankovič et al. 2022; Wang et al. 2021].

Higher Moments for PP Analysis. Using concentration-of-measure inequalities [Boucheron et al. 2013], we can utilize higher moments $\mathbb{E}(X^k)$ to obtain upper and lower bounds on tail probabilities $\mathbb{P}(X > t)$, measuring the probability that a given random variable X , corresponding for example to our program variables x from Figure 1, surpasses some value t . *In this paper, we also show that when a program variable x admits only $k < \infty$ many values, we can fully recover its probability mass function as a closed-form expression in the loop counter n using the first $k-1$ raw moments* (see Section 6). Furthermore, raw moments can be used to compute central moments $\mathbb{E}((X - \mathbb{E}(X))^k)$ and thus provide insights on other important characteristics of the distribution such as the *variance*, *skewness* and *kurtosis* [Durrett 2019]. However, *computing exact higher statistical moments* for PPs is computationally expensive [Kaminski et al. 2019], a challenge which we also *address in this paper*, as illustrated in Figures 1–2 and described next.

Computing Higher Moments. The theory we establish in this paper describes how to compute higher moments of program variables for a large class of probabilistic loops and how to utilize these moments to gain more insights into the analyzed programs. We call this theory the *theory of moment-computable probabilistic loops* (Section 5). Our approach is fully automatic, meaning it does *not* rely on externally provided invariants or templates. Unlike constraint solving over templates [Barthe et al. 2016; Kura et al. 2019], we employ algebraic techniques based on systems of *linear recurrences* with constant coefficients describing so-called *C-finite sequences* [Kauers and Paule 2011]. Different equivalence preserving program transformations (Section 3) and power reduction of finite valued variables (Section 4) allow us to simplify PPs and represent their higher moments as linear recurrence systems in the loop counter. Figure 1 shows a PP with many unique features supported by our work towards PP analysis: it has an uncountable state-space, contains if-statements, symbolic constants, draws from continuous probability distributions with state-dependent parameters, and employs polynomial arithmetic as well as circular variable dependencies. *We are not aware of other works automating the reasoning about such and similar probabilistic loops*, in particular for computing precise higher moments of variables. Figure 1 lists some of the variables’

```

toggle, sum, x, y, z = 0, s0, 1, 1, 1
while ★:
    toggle = 1 - toggle
    if toggle == 0:
        x = x + 1 {1/2} x + 2
        y = y + z + x ** 2 {1/3} y - z - x
        z = z + y {1/4} z - y
    end
    l, g = Laplace(x + y, 1), Normal(0, 1)
    if g < 1/2: sum = sum + x end
end

```

$$\begin{aligned}
\mathbb{E}(\text{toggle}_n) &= \frac{1}{2} - \frac{(-1)^n}{2} \\
\mathbb{E}(x_n) &= \frac{5}{8} + \frac{3n}{4} + \frac{3(-1)^n}{8} \\
\mathbb{E}(x_n^2) &= \frac{15}{32} + \frac{17n}{16} + \frac{9n(-1)^n}{16} + \frac{17(-1)^n}{32} + \frac{9n^2}{16} \\
\mathbb{E}(l_n) &= \frac{-17}{8} - \frac{15n}{4} + \frac{67 \cdot 2^{-n} \cdot 6^{\frac{n}{2}}}{10} + \frac{67 \cdot 2^{-n} 6^{\frac{1+n}{2}}}{30} - \frac{37 \cdot 3^{-n} 6^{\frac{n}{2}}}{10} - \frac{37 \cdot 3^{-n} 6^{\frac{1+n}{2}}}{20} + \frac{67 \cdot 6^{\frac{n}{2}} (-1)^n}{10 \cdot 2^n} - \frac{15(-1)^n}{8} - \frac{67 \cdot 6^{\frac{1+n}{2}} (-1)^n}{30 \cdot 2^n} + \frac{37 \cdot 6^{\frac{1+n}{2}} (-1)^n}{20 \cdot 3^n} - \frac{37 \cdot 6^{\frac{n}{2}} (-1)^n}{10 \cdot 3^n}
\end{aligned}$$

Fig. 1. An example of a multi-path PP loop, with Laplace and Normal distributions parametrized by program variables. Our work fully automates the analysis of such and similar PP loops by computing higher moments. Several moments for program variables in the loop counter n are listed on the bottom, each moment was automatically generated.

moments computed automatically by our work. Further, these moments can be used to compute tail probability bounds or central moments, such as the variance, to characterize the distribution of the program variables as the loop progresses.

Thanks to our power reduction techniques (Section 4), our approach supports arbitrary polynomial dependencies among finite valued variables. Moreover, our work can fully recover the value distributions of finite valued program variables, from finitely many higher moments, as illustrated in Section 6 for Herman’s self-stabilization algorithm from Figure 2.

Theory and Practice in Computing Higher Moments. In theory, our approach can compute any higher moment for any variable and PP of our program model, under assumptions stated in Sections 3 and 5. We also establish the necessity of these assumptions in Section 5.3. In a nutshell, the completeness theorem (Theorem 6) holds for probabilistic loops for which non-finite program variables are not polynomially self-dependent and all branching conditions are over finite valued variables. We strengthen the theory of [Bartocci et al. 2019] to support if-statements, circular variable dependencies, state-dependent distribution parameters, simultaneous assignments, and multiple assignments, and establish the necessity of our assumptions. Moreover, unlike [Wang et al. 2021], our approach does not rely on templates and provides exact closed-form representations of higher moments parameterized by the loop counter.

In practice, our approach is implemented in the Polar tool and compared against exact as well as approximate methods. Our experiments (Section 7) show that Polar outperforms the state-of-the-art of moment computation for probabilistic loops in terms of supported programs and efficiency. Furthermore, Polar is able to compute exact higher moments magnitudes faster than sampling can establish reasonable confidence intervals.

Contributions. Our main contributions are listed below:

- An automated approach for computing higher moments of program variables for a large class of probabilistic loops with potentially uncountable state spaces (Sections 3-5).
- We develop power reduction techniques to reduce the degrees of finite valued program variables in polynomials (Section 4).

```

x1, x2, x3 = 1, 1, 1
t1, t2, t3 = 1, 1, 1
p = 1/2; tokens = t1 + t2 + t3
while ★:
    x1o, x2o, x3o = x1, x2, x3
    if x1o == x3o: x1=Bernoulli(p) else: x1=x3o end
    if x2o == x3o: x2=Bernoulli(p) else: x2=x1o end
    if x3o == x2o: x3=Bernoulli(p) else: x1=x3o end

    if x1 == x3: t1 = 1 else: t1 = 0
    if x2 == x1: t2 = 1 else: t2 = 0
    if x3 == x2: t3 = 1 else: t3 = 0
    tokens = t1 + t2 + t3
end

```

$$\mathbb{E}(\text{tokens}_n) = 1 + 2 \cdot 4^{-n} \quad | \quad \mathbb{E}(\text{tokens}_n^2) = 1 + 8 \cdot 4^{-n} \quad | \quad \mathbb{E}(\text{tokens}_n^3) = 1 + 26 \cdot 4^{-n}$$

Fig. 2. Herman’s self stabilization algorithm with three nodes encoded as a probabilistic loop together with three moments of *tokens*.

- We prove completeness of our work for computing higher moments (Section 5).
- We fully recover the distributions of finite valued program variables and approximate distributions for unbounded/continuous program variables from finitely many moments (Section 6).
- We provide an implementation and empirical evaluation of our work, outperforming the state-of-the-art in PP analysis in terms of automation and expressivity (Section 7).

2 PRELIMINARIES

We use the symbol \mathbb{P} for probability measures and \mathbb{E} for the expectation operator. The support of a random variable X is denoted by $\text{supp}(X)$.

2.1 Probability Theory

Operationally, a probabilistic program is a Markov chain with potentially uncountably many states. Let us recall some notions about Markov chains. For more details on Markov chains and probability theory in general we refer the reader to [Durrett 2019].

DEFINITION 1 (SEQUENCE SPACE). *Let (S, \mathcal{S}) be a measurable space. Its sequence space is the measurable space $(S^\omega, \mathcal{S}^\omega)$ where $S^\omega := \{(s_1, s_2, \dots) : s_i \in S\} = \{\text{functions } \theta : \mathbb{N} \rightarrow S\}$ and \mathcal{S}^ω is the σ -algebra generated by the sets $\{\theta : \theta_i \in B_i, 1 \leq i \leq n\}$ where $B_i \in \mathcal{S}$.*

A *Markov kernel* is, on a high level, a generalization of transition probabilities between states to uncountable state spaces and is required for the definition of a *Markov chain*.

DEFINITION 2 (MARKOV CHAIN). *Let $(S, \mathcal{S}, \mathbb{P})$ be a probability space and $p : S \times S \rightarrow [0, 1]$ a Markov kernel. A stochastic process X_n is a Markov chain with Markov kernel p if*

$$\mathbb{P}(X_{n+1} \in B \mid X_0, X_1, \dots, X_n) = p(X_n, B). \quad (1)$$

Given a measurable space (S, \mathcal{S}) , an *initial distribution* μ , a stochastic process X_n and a Markov kernel p , Kolmogorov’s *Extension Theorem* says that there is a unique measure \mathbb{P} such that X_n is a Markov chain in $(S^\omega, \mathcal{S}^\omega, \mathbb{P})$.

```

 $\text{lop} \in \{\text{and}, \text{or}\}, \text{cop} \in \{=, \neq, <, >, \geq, \leq\}, \text{Dist} \in \{\text{Bernoulli}, \text{Normal}, \text{Uniform}, \dots\}$ 
 $\langle \text{sym} \rangle ::= a \mid b \mid \dots \mid \langle \text{var} \rangle ::= x \mid y \mid \dots$ 
 $\langle \text{const} \rangle ::= r \in \mathbb{R} \mid \langle \text{sym} \rangle \mid \langle \text{const} \rangle (+ \mid * \mid /) \mid \langle \text{const} \rangle$ 
 $\langle \text{poly} \rangle ::= \langle \text{const} \rangle \mid \langle \text{var} \rangle \mid \langle \text{poly} \rangle (+ \mid - \mid *) \mid \langle \text{poly} \rangle ** n$ 
 $\langle \text{assign} \rangle ::= \langle \text{var} \rangle = \langle \text{assign\_right} \rangle \mid \langle \text{var} \rangle, \langle \text{assign} \rangle, \langle \text{assign\_right} \rangle$ 
 $\langle \text{categorical} \rangle ::= \langle \text{poly} \rangle \{ \langle \text{const} \rangle \} \langle \text{poly} \rangle^* [ \{ \langle \text{const} \rangle \} ]$ 
 $\langle \text{assign\_right} \rangle ::= \langle \text{categorical} \rangle \mid \text{Dist}(\langle \text{poly} \rangle^*) \mid \text{Exponential}(\langle \text{const} \rangle / \langle \text{poly} \rangle)$ 
 $\langle \text{bexpr} \rangle ::= \text{true} (\star) \mid \text{false} \mid \langle \text{poly} \rangle \langle \text{cop} \rangle \langle \text{poly} \rangle \mid \text{not} \langle \text{bexpr} \rangle \mid \langle \text{bexpr} \rangle \langle \text{lop} \rangle \langle \text{bexpr} \rangle$ 
 $\langle \text{ifstmt} \rangle ::= \text{if} \langle \text{bexpr} \rangle : \langle \text{stmts} \rangle (\text{else if} \langle \text{bexpr} \rangle : \langle \text{stmts} \rangle)^* [\text{else} : \langle \text{stmts} \rangle] \text{end}$ 
 $\langle \text{stmt} \rangle ::= \langle \text{assign} \rangle \mid \langle \text{ifstmt} \rangle \quad \langle \text{stmts} \rangle ::= \langle \text{stmt} \rangle^+$ 
 $\langle \text{loop} \rangle ::= \langle \text{stmt} \rangle^* \text{while} \langle \text{bexpr} \rangle : \langle \text{stmts} \rangle \text{end}$ 

```

Fig. 3. Grammar describing the syntax of probabilistic loops $\langle \text{loop} \rangle$.

For a random variable X , central moments $\mathbb{E}((X - \mathbb{E}(X))^k)$ can be computed from raw moments $\mathbb{E}(X^k)$ and vice versa through the transformation of center:

$$\mathbb{E}((X - b)^k) = \sum_{i=0}^k \binom{k}{i} \mathbb{E}((X - a)^i) (a - b)^{k-i}. \quad (2)$$

2.2 Linear Recurrences

We briefly recall standard terminology on algebraic sequences and recurrences. For further details, we refer the reader to [Kauers and Paule 2011]. A sequence $(a_n)_{n=0}^\infty$ is called *C-finite* if it obeys a linear recurrence with constant coefficients.

Theorem 1 (Closed-form). *Every C-finite sequence $(a_n)_{n=0}^\infty$ can be written as an exponential polynomial, that is $a_n = \sum_{i=1}^m n^{d_i} u_i^n$ for some $d_i \in \mathbb{N}$ and $u_i \in \mathbb{C}$. We refer to $\sum_{i=1}^m n^{d_i} u_i^n$ as the closed-form or the solution of the sequence $(a_n)_{n=0}^\infty$ or its recurrence.*

An important fact is that closed-forms of any component of systems of linear recurrences with constant coefficients of *any order* are computable.

3 PROBABILISTIC PROGRAM MODEL

In this section we introduce our programming model (Section 3.1) and describe its semantics in terms of Markov chains (Section 3.2). Moreover, we introduce transformations (3.3) normalizing a probabilistic program to simplify its analysis.

3.1 Probabilistic Program Syntax

The syntax defining our program model is given by the grammar of Figure 3. Throughout the paper, we will use the phrases *(probabilistic) loops* and *(probabilistic) programs* interchangeably for loops adhering to the syntax in Figure 3. In our work, we infer higher moments $\mathbb{E}(x_n^k)$ of program variables x parameterized by the loop counter n . We abstract from concrete loop guards by defining the guards of programs in our program model to be *true* (written as \star). Guarded loops **while** $\phi : \dots$ can be modeled as an infinite loops **while** \star : **if** $\phi : \dots$, with the limit behaviour giving the moments after termination (cf. Section 5.1).

Our program model defined in Figure 3 contains non-nested while-loops which are preceded by a loop-free initialization part. The loop-body and initialization part allow for (nested) if-statements, polynomial arithmetic, drawing from common probability distributions, and symbolic constants.

Symbolic constants can be used to represent arbitrary real numbers and are also used for uninitialized program variables. Categorical expressions (defined by the non-terminal $\langle \text{categorical} \rangle$ in Figure 3) are expressions of the form $v_1\{p_1\} \dots v_l\{p_l\}$ such that $\sum p_i = 1$. Their intended meaning is that they evaluate to v_i with probability p_i . The last parameter p_l can be omitted and in that case is set to $p_l := 1 - \sum_{i=1}^{l-1} p_i$. For a program \mathcal{P} we denote with $\text{Vars}(\mathcal{P})$ the set of \mathcal{P} 's variables appearing on the left-hand side of an assignment in \mathcal{P} 's loop-body. The programs of Figures 1-2 are examples of our program model defined in Figure 3. In comparison to the *probabilistic Guarded Command Language* (pGCL) [Barthe et al. 2020], programs of our model contain exactly one while-loop, no non-determinism but support continuous distributions and simultaneous assignments.

3.2 Program Semantics

In what follows we define the semantics of probabilistic programs in terms of Markov chains on a measurable space. We then introduce the notion of *normalized* probabilistic loops by means of so-called \mathcal{P} -preserving program transformations (Section 3.3).

DEFINITION 3 (STATE & RUN SPACE). *Let \mathcal{P} be a probabilistic program with m variables. We denote by $\text{ND}(\mathcal{P})$ the non-probabilistic program obtained from \mathcal{P} by replacing every probabilistic choice C in \mathcal{P} by a non-deterministic choice over $\text{supp}(C)$. Let $\text{States}_{\mathcal{P}} \subseteq \mathbb{R}^m$ be the set of program states of $\text{ND}(\mathcal{P})$ reachable from any initial state. The state space of \mathcal{P} is the measurable space $(\text{States}_{\mathcal{P}}, \mathcal{S}_{\mathcal{P}})$, where $\mathcal{S}_{\mathcal{P}}$ is the Borel σ -algebra on \mathbb{R}^m restricted to $\text{States}_{\mathcal{P}}$. The run space of \mathcal{P} is the sequence space $(\text{States}^{\omega}, \mathcal{S}^{\omega}) =: (\text{Runs}_{\mathcal{P}}, \mathcal{R}_{\mathcal{P}})$.*

In what follows, we omit the subscript \mathcal{P} whenever the program \mathcal{P} is irrelevant or clear from the context. Executions/runs of a probabilistic program \mathcal{P} define a stochastic process, as follows.

DEFINITION 4 (RUN PROCESS). *Let \mathcal{P} be a probabilistic program with m variables. The run process $\Phi_n : \text{Runs} \rightarrow \text{States}$ is a stochastic process in the run space mapping a program run to its n th state, that means, $\Phi_n(\text{run}) := \text{run}_n$.*

For program variable x with index $i \geq 1$, we denote by x_n and $\Phi_n(\cdot)(x)$ the projection of Φ_n to its i th component $\Phi_n(\cdot)(i)$. Given an arithmetic expression A over \mathcal{P} 's variables, we write A_n for the stochastic process where every program variable x in A is replaced by x_n .

Remark. *Given an initial distribution of program states μ and a Markov kernel p defined according to the standard meaning of the program statements, by Kolmogorov's Extension Theorem we conclude that there is a unique probability measure $\mathbb{P}_{\mathcal{P}}$ on $(\text{Runs}_{\mathcal{P}}, \mathcal{R}_{\mathcal{P}})$ such that X_n is a Markov chain. $(\text{Runs}_{\mathcal{P}}, \mathcal{R}_{\mathcal{P}}, \mathbb{P}_{\mathcal{P}})$ is the probability space associated to program \mathcal{P} . (Higher) moments of \mathcal{P} 's variables are to be understood with respect to this probability space.*

For probabilistic loops according to the syntax in Figure 3, the initial distribution μ of values of loop variables is the distribution of states after the statements $\langle \text{statement} \rangle^*$ just before the while-loop. Moreover, the loop body in Figure 3 is considered to be atomic, meaning the Markov kernel p describes the transition between full iterations in contrast to single statements.

3.3 \mathcal{P} -Preserving Transformations

Our probabilistic programs defined by the grammar in Figure 3 support rich arithmetic and complex probabilistic behavior/distributions. Such an expressivity of Figure 3 comes at the cost of turning the analysis of programs defined by Figure 3 cumbersome. In this section, we address this difficulty and introduce a number of program transformations that allow us to simplify our probabilistic programs to a so-called *normal form* while preserving all (relevant) program properties. Normal forms allow us to extract recursive properties from the program, which we will later use to compute moments for program variables (Section 5).

Schemas and Unification. The program transformations we introduce in this section build on the notion of *schemas* and program parts. A *program part* is an empty word or any word resulting from any non-terminal of the grammar in Figure 3. For our purposes, a schema S is a program part with some subtrees in the program part's syntax tree being replaced by placeholder symbols s_1, \dots, s_l . A *substitution* is a finite mapping $\sigma = \{s_1 \mapsto p_1, \dots, s_l \mapsto p_l\}$ where p_1, \dots, p_l are program parts. We denote by $S[\sigma]$ the program part resulting from S by replacing every s_i by p_i , assuming $S[\sigma]$ is well-formed. For two schemas S_1 and S_2 a substitution u such that $S_1[u] = S_2[u]$ is called a *unifier* (with respect to S_1 and S_2). In this case S_1 and S_2 are called *unifiable* (by u).

Transformations. In what follows, we consider \mathcal{P} to be a fixed probabilistic program defined by Figure 3 and give all definitions relative to \mathcal{P} . A *transformation* T is a mapping from program parts to program parts with respect to a schema *Old*. T is *applicable* to a subprogram S of \mathcal{P} if S and *Old* are unifiable by the unifier u . Then, the transformed subprogram is defined as $T(S) := \text{New}[u]$ where *New* is a schema depending on *Old* and u . A transformation is fully specified by defining how *New* results from *Old* and u . We write $T(\mathcal{P}, S)$ for the program resulting from \mathcal{P} by replacing the subprogram S of \mathcal{P} by $T(S)$.

The first transformation we consider removes simultaneous assignments from \mathcal{P} . For this, we store a copy for each of the assignments in an auxiliary variable to preserve values used for simultaneous assignments. Variables are then assigned their intended value.

DEFINITION 5 (SIMULTANEOUS ASSIGNMENT TRANSFORMATION). A simultaneous assignment transformation is the transformation defined by

$$x_1, \dots, x_l = v_1, \dots, v_l \mapsto t_1 = v_1; \dots; t_l = v_l; \dot{x}_1 = t_1; \dots; \dot{x}_l = t_l,$$

where t_1, \dots, t_l are fresh variables.

In what follows, we assume that parameters of common distributions used in programs are constant. Nevertheless, the following transformation enables the use of some non-constant distribution parameters.

DEFINITION 6 (DISTRIBUTION TRANSFORMATION). A distribution transformation is a transformation defined by either of the mappings

- $\dot{x} = \text{Normal}(\dot{p}, \dot{v}) \mapsto t = \text{Normal}(0, \dot{v}); \dot{x} = \dot{p} + t$
- $\dot{x} = \text{Uniform}(p_1, p_2) \mapsto t = \text{Uniform}(0, 1); \dot{x} = \dot{p}_1 + (p_2 - p_1) * t$
- $\dot{x} = \text{Laplace}(\dot{p}, \dot{b}) \mapsto t = \text{Laplace}(0, \dot{b}); \dot{x} = \dot{p} + t$
- $\dot{x} = \text{Exponential}(\dot{c}/\dot{p}) \mapsto t = \text{Exponential}(\dot{c}); \dot{x} = \dot{p} * t$

where, for every mapping, t is a fresh variable.

Example 1. Consider Figure 1. The simultaneous assignment $\text{lg} = \text{Laplace}(x+y, 1), \text{Normal}(0, 1)$ can be transformed using the transformation rules from Definitions 5-6 as follows:

$$\begin{array}{ccc}
 & t1 = \text{Laplace}(x+y, 1) & t3 = \text{Laplace}(0, 1) \\
 & t2 = \text{Normal}(0, 1) & t1 = x+y+t3 \\
 \text{(sim)} \mapsto & x = t1 & \text{(dist)} \mapsto t2 = \text{Normal}(0, 1) \\
 & y = t2 & x = t1 \\
 & & y = t2
 \end{array}$$

To simplify the structure of probabilistic loops, we assume **else if** branches to be syntactic sugar for nested **if else** statements. We remove **else** by splitting it into if-statements (**if** C and **if not** C). Since variables in C could be changed within the first branch, we store their original values in auxiliary variables and use those for the condition C' of the second **if** statement. We capture this transformation in the following definition.

DEFINITION 7 (ELSE TRANSFORMATION). An else transformation is the transformation

$$\begin{array}{lcl} \text{if } \dot{C} : \text{Br}\ddot{a}\text{nc}h_1 & , u \mapsto & t_1 = x_1 ; \dots ; t_l = x_l \\ \text{else} : \text{Br}\ddot{a}\text{nc}h_2 \text{ end} & & \text{if } \dot{C} : \text{Br}\ddot{a}\text{nc}h_1 \text{ end} \\ & & \text{if not } C' : \text{Br}\ddot{a}\text{nc}h_2 \text{ end} \end{array}$$

where x_1, \dots, x_l are all variables appearing in $\dot{C}[u]$ which are also being assigned in $\text{Br}\ddot{a}\text{nc}h_1[u]$. Every t_i is a fresh variable and C' results from $\dot{C}[u]$ by substituting every x_i with t_i .

To further simplify the loop body into a flattened list of assignments, we equip every assignment a of form “ $x = \text{value}$ ” with a condition C_a (initialized to true \top) and a default variable d_a (initialized to x), written as “ $x = \text{value} [C_a] d_a$ ”. The semantics of the conditioned assignment is that x is assigned *value* if C_a holds just before the assignment and d_a otherwise. With conditioned assignments, the loop body’s structure can be flattened using the following transformation.

DEFINITION 8 (IF TRANSFORMATION). An if transformation is the transformation defined by

$$\begin{array}{lcl} \text{if } \dot{C}_1 : & t = \dot{x} & \\ \dot{x} = \dot{v} [\dot{C}_2] \dot{x} & , u \mapsto & \dot{x} = \dot{v} [\dot{C}_1 \text{ and } \dot{C}_2] \dot{x} \\ \text{Rest end} & & \text{if } C : \text{Rest end} \end{array}$$

where t is a fresh variable and C results from $\dot{C}_1[u]$ by substituting $\dot{x}[u]$ by t . If $\text{Rest}[u]$ is empty, the line **if** $C : \text{Rest}$ **end** is omitted from the result. If $\dot{x}[u]$ does not appear in $\dot{C}_1[u]$ the line $t = \dot{x}$ is dropped.

To bring further simplicity to our program \mathcal{P} , we ensure for each variable to be modified only once within the loop body. To remove duplicate assignments we introduce new variables x_1, \dots, x_{l-1} to store intermediate states. Assignments to other variables, in between the updates of x , will be adjusted to refer to the latest x_i instead of x .

DEFINITION 9 (MULTI-ASSIGNMENT TRANSFORMATION). A multi-assignment transformation is the transformation defined by

$$\begin{array}{lcl} \dot{x} = v_1 [\dot{C}_1] \dot{x} ; \text{Rest}_1 ; & & x_1 = v_1 [\dot{C}_1] \dot{x} ; \text{Rest}_1 ; \\ \dot{x} = v_2 [\dot{C}_2] \dot{x} ; \text{Rest}_2 ; & , u \mapsto & x_2 = v_2 [\dot{C}_2] x_1 ; \text{Rest}_2 ; \\ \dots ; \dot{x} = v_l [\dot{C}_l] \dot{x} ; & & \dots ; \dot{x} = v_l [\dot{C}_l] x_{l-1} ; \end{array}$$

where x_1, \dots, x_{l-1} are fresh variables. For $i \geq 2$, v_i , C_i and Rest_i result from $v_i[u]$, $\dot{C}_i[u]$ and $\text{Rest}_i[u]$, respectively, by replacing $\dot{x}[u]$ by x_{i-1} .

Example 2. In the program of Figure 2, program line

if $x_1 == x_3$: token1 = 1 **else** : token1 = 0 can be transformed using transformation rules from Definitions 7-9 as follows:

$$\begin{array}{lcl} \text{(else)} & & \text{if } x_1 == x_3 : t_1 = 1 \\ \mapsto & & \text{if } x_1 != x_3 : t_1 = 0 \\ \\ \text{(if)} & & t_1 = 1 [x_1 == x_3] t_1 \\ \mapsto & & t_1 = 0 [x_1 != x_3] t_1 \\ \\ \text{(multi)} & & t_{11} = 1 [x_1 == x_3] t_1 \\ \mapsto & & t_1 = 0 [x_1 != x_3] t_{11} \end{array}$$

With program transformations defined, we can turn our attention to program properties. In particular, we show that our transformations of \mathcal{P} do not change statistical properties of \mathcal{P} . Since our transformations may introduce new variables, we consider program equivalence with respect to program variables in order to ensure that properties of \mathcal{P} are maintained/preserved by our transformations.

DEFINITION 10 (PROGRAM EQUIVALENCE). Let \mathcal{P}_1 and \mathcal{P}_2 be two probabilistic programs with run processes Φ_n and Ψ_n , respectively. We define \mathcal{P}_1 and \mathcal{P}_2 to be equivalent with respect to program variable x , in symbols $\mathcal{P}_1 \equiv^x \mathcal{P}_2$, if:

- (1) $x \in \text{Vars}(\mathcal{P}_1) \cap \text{Vars}(\mathcal{P}_2)$, and
- (2) $\Phi_n(\cdot)(x) = \Psi_n(\cdot)(x)$ for all $n \in \mathbb{N}$.

To relate a program \mathcal{P} to its transformed version $T(\mathcal{P}, S)$ we consider properties of variables of the original program \mathcal{P} . If all variables of \mathcal{P} retain their properties after applying transformation T , we say that T is \mathcal{P} -preserving.

DEFINITION 11 (\mathcal{P} -PRESERVING TRANSFORMATION). We say T is \mathcal{P} -preserving if $\mathcal{P} \equiv^x T(\mathcal{P}, S)$ for all $x \in \text{Vars}(\mathcal{P})$ and all subprograms S of \mathcal{P} which are unifiable with Old .

It is not hard to argue that the transformations defined above are \mathcal{P} -preserving, yielding the following result.

Lemma 2. *The transformations from Definitions 5-9 are \mathcal{P} -preserving.*

By exhaustively applying the \mathcal{P} -preserving transformations of Definitions 5-9 over \mathcal{P} , we obtain a so-called normalized program \mathcal{P}_N , as defined below. The normalized \mathcal{P}_N will then further be used in computing higher moments of \mathcal{P} in Sections 5, as the \mathcal{P}_N preserves the moments of \mathcal{P} (Theorem 3).

DEFINITION 12 (NORMAL FORM). A program \mathcal{P} is in normal form or a normalized program if none of the transformations from Definitions 5-9 are applicable to \mathcal{P} .

Theorem 3 (Normal Form). *For every probabilistic program \mathcal{P} there is a \mathcal{P}_N in normal form such that $\mathcal{P} \equiv^x \mathcal{P}_N$ for all $x \in \text{Vars}(\mathcal{P})$. Moreover, \mathcal{P}_N can be effectively computed from \mathcal{P} by exhaustively applying transformations from Definitions 5-9.*

PROOF. There are two claims in the theorem, which we need to address: (i) exhaustively applying transformation rules terminates (*termination*), and (ii) it preserves statistical properties of the (original) program variables (*correctness*).

For termination, we show that programs become smaller, in some sense, after every transformation. In particular, we consider the program size to be given by a tuple $(\text{Sim}, \text{Dist}, \text{Else}, \text{If}, \text{Multi}_B, \text{Multi}_I)$, representing the number of simultaneous assignments, non-trivial distributions, else statements, assignments within if branches (weighted for nested ifs), and number of variables with multiple assignments in the loop body and initialization part, respectively. With respect to the lexicographic order, each transformation reduces the size of the program which is lower-bounded by 0.

Correctness can be shown by treating each transformation separately and showing that it does not alter the variables' distributions after a single application. This is true for all transformations from Definitions 5-9. Auxiliary variables are used to store the original value to prevent intervening variable modifications. For Multi-Assignment Transformation (Definition 9), we also revise the rest of the assignments to reflect the change of the original variable. The Distribution Transformation (Definition 6) uses statistical properties of well-known distributions. \square

Remark. *Based on the order in which transformations are applied to a program \mathcal{P} and the names used for auxiliary variables, several different normalized programs can be achieved for \mathcal{P} . In this work, only the existence of a normal form is relevant. Moreover, from the definitions of the transformation, it is apparent that exhaustively applying them leads to a normal form whose size is linear in the size of the original program.*

```

toggle = 0; sum = s0
x = 1; y = 1; z = 1
while ★:
    toggle = 1 - toggle
    x = 1+x {1/2} 2+x [toggle==0] x
    y = y+z+x**2 {1/3} -x+y-z [toggle==0] y
    z = z+y {1/4} z-y [toggle==0] z
    t1 = Laplace(0, 1)
    l = t1+x+y
    g = Normal(0, 1)
    sum = sum+x [g < 1/2] sum
end

```

Fig. 4. A normal form for the program in Figure 1.

4 FINITE TYPES IN PROBABILISTIC PROGRAMS

Given a probabilistic program \mathcal{P} in our programming model, the transformations of Section 3.3 simplify \mathcal{P} by computing its normalized form while maintaining statistical properties (moments among others) of \mathcal{P} . Nevertheless, the normalized form of \mathcal{P} implements computationally expensive polynomial arithmetic, potentially hindering the automated analysis of \mathcal{P} in Section 5. Therefore, we introduce further simplifications for \mathcal{P} by means of power reduction techniques.

Example 3. Consider Figure 1 and assume we are interested in the k th power of variable `toggle` and deriving the raw moment $\mathbb{E}(\text{toggle}_n^k)$. Our analysis relies on replacing variables with their assignments (see Section 5), leading to expression $(1 - \text{toggle}_{n-1})^k$. When expanded, this is a polynomial in toggle_{n-1} with $k + 1$ monomials. Higher moments, together with the aforementioned replacements, may lead to blowups of the number of monomials to consider. However, observing that the variable `toggle` is binary, we have $\text{toggle}_n^k = \text{toggle}_n = 1 - \text{toggle}_{n-1}$ for any $k \geq 0$. Arbitrary powers of finite variable `toggle` with 2 possible values can be written in terms of powers smaller than 2. In the rest of this section, we show that this phenomenon generalizes from binary variables to arbitrary finite valued variables, thus simplifying the analysis of higher moments of finite valued program variables.

DEFINITION 13 (FINITE EXPRESSION). Let \mathcal{P} be a probabilistic program and A an arithmetic expression over the variables of \mathcal{P} . We say that A is finite if there exist $a_1, \dots, a_m \in \mathbb{R}$ such that for all $n \in \mathbb{N} : A_n \in \{a_1, \dots, a_m\}$.

4.1 Power Reduction for Finite Types

As established in [Bartocci et al. 2020a], high powers k of a random variable X over a finite set can be reduced. Adapted to our setting, we obtain the following result.

Theorem 4 (Finite Power Reduction). Let $m, k \in \mathbb{Z}$ and X be a discrete random variable over $A = \{a_1, \dots, a_m\}$. Then we can rewrite X^k as a linear combination of $1, X, X^2, \dots, X^{m-1}$. Furthermore,

$$X^k = \overline{a^k} M^{-1} \overline{X}, \quad (3)$$

where $\overline{a^k} = (a_1^k, \dots, a_m^k)$, M is an $m \times m$ matrix with $M_{ij} = a_j^{i-1}$ (with $0^0 := 1$), and $\overline{X} = (X^0, \dots, X^{m-1})^T$.

In other words, Theorem 4 implies that any higher moment of X , can be computed from just its first $m-1$ moments. Furthermore, we build on Theorem 4 and establish the inverse of matrix M explicitly.

Theorem 5 (Reduction Formula). *Recall that the k th elementary symmetric polynomial with respect to a set $V = \{v_1, \dots, v_n\}$ is $e_k(V) = \sum_{1 \leq j_1 < \dots < j_k \leq n} v_{j_1} \cdots v_{j_k}$ and let $A_{-j} = A \setminus \{a_j\}$. Then the inverse of M in (3) is given by*

$$M_{ij}^{-1} = -\frac{(-1)^j e_{m-j}(A_{-i})}{\prod_{a \in A_{-i}} (a - a_i)}. \quad (4)$$

PROOF. Let $MN = B$ for M as of (3) and N as of (4). We show that $B = I$ by showing that $B_{ij} = 1$ iff $i = j$ and $B_{ij} = 0$ otherwise. We have

$$\begin{aligned} B_{ij} &= \sum_{1 \leq k \leq m} N_{ik} M_{kj} = \sum_{1 \leq k \leq m} -\frac{(-1)^k e_{m-k}(A_{-i})}{\prod_{a \in A_{-i}} (a - a_i)} a_j^{k-1} \\ &= \frac{1}{\prod_{a \in A_{-i}} (a - a_i)} \sum_{1 \leq k \leq m} -(-1)^k a_j^{k-1} e_{m-k}(A_{-i}) \\ &= \frac{1}{\prod_{a \in A_{-i}} (a - a_i)} \sum_{0 \leq k \leq m-1} (-a_j)^k e_{m-k+1}(A_{-i}) \\ &= \frac{1}{\prod_{a \in A_{-i}} (a - a_i)} \prod_{a \in A_{-j}} (a - a_i), \end{aligned}$$

where the last equation comes from the expansion of product $\prod_{a \in A_{-j}} (a - a_i)$ and grouping by the exponent of a_j . We can clearly see that the last expression is 1 if $i = j$ and 0 otherwise. \square

Example 4. Let X be a random variable over $A := \{-2, 0, 1, 3\}$. Using Theorem 4-5, we obtain the 10th power of X as:

$$\begin{aligned} X^{10} &= \begin{pmatrix} (-2)^{10} & 0^{10} & 1^{10} & 3^{10} \end{pmatrix} \begin{pmatrix} 0 & -1/10 & 2/5 & -1/30 \\ 1 & -5/6 & -1/3 & 1/6 \\ 0 & 1 & 1/6 & -1/6 \\ 0 & -1/15 & 1/30 & 1/30 \end{pmatrix} \begin{pmatrix} X^0 \\ X^1 \\ X^2 \\ X^3 \end{pmatrix} \\ &= 1934X^3 + 2105X^2 - 4038X. \end{aligned}$$

5 COMPUTING HIGHER MOMENTS OF PROBABILISTIC PROGRAMS

We now bring together the results from Sections 3-4 to develop the *theory of moment-computability* for probabilistic loops. We establish the technical details leading to sufficient conditions that ensure moment-computability, culminating in the proof of Theorem 6. The main ideas of our method are illustrated on the probabilistic loop from Figure 1 in Example 5 at the end of this section.

DEFINITION 14 (MOMENT-COMPUTABILITY). *A probabilistic loop \mathcal{P} is moment-computable if a closed-form (according to Theorem 1) of $\mathbb{E}(x_n^k)$ exists and is computable for all $x \in \text{Vars}(\mathcal{P})$ and $k \in \mathbb{N}$.*

We will describe the class of moment-computable probabilistic loops through the properties of the dependencies between program variables.

DEFINITION 15 (VARIABLE DEPENDENCY). *Let \mathcal{P} be a probabilistic loop and $x, y \in \text{Vars}(\mathcal{P})$. We define:*

- y depends conditionally on x , if there is an assignment of y within an if-else-statement and x appears in the if-condition.
- y depends finitely on x , if x is finite and appears in an assignment of y .
- y depends linearly on x , if x appears only linearly in every assignment of y .

- y depends polynomially on x , if there is an assignment of y in which x appears non-linearly and x is not finite (motivated by Section 4.1).
- y depends on x if it depends on x conditionally, finitely, linearly, or polynomially.

Furthermore, we consider the transitive closure for variable dependency as follows: If z depends on y and y depends on x , then z depends on x . If one of the two dependencies is polynomial, then z depends polynomially on x .

A crucial point to highlight in Definition 15 is that due to transitivity, variables can depend on themselves. For instance, if variable x depends on y and y on x , then x is self-dependent. Moreover, if either of the dependencies between x and y is non-linear, x depends, by Definition 15, *polynomially on itself*. The absence of such polynomial self-dependencies is a central condition for our notion of moment-computable loops.

Theorem 6 (Moment-Computability). *A probabilistic loop \mathcal{P} is moment-computable if (1) none of its non-finite variables depends on itself polynomially, and (2) if the variables in all if-conditions are finite.*

Note that none of the program transformations from Section 3.3 can introduce a polynomial (self-)dependence. We capture this in the following lemma:

Lemma 7 (Non-Dependency Preservation). *If a variable $x \in \text{Vars}(\mathcal{P})$ does not depend on itself polynomially, neither does $x \in \text{Vars}(\mathcal{P}_N)$.*

Before we prove Theorem 6, let us first show its validity for programs in normal form (as defined in Section 3.3). Recall that a normalized program's loop body is a flat list of (guarded) assignments, one for every (possibly auxiliary) program variable.

Lemma 8 (Normal Moment-Computability). *The Moment-Computability Theorem (Theorem 6) holds for loops in normal form.*

PROOF. The main idea of the proof is to show that for any monomial M , the moment $\mathbb{E}(M)$ only depends on a finite set of monomials, each (in some sense) not *larger* than M itself. Furthermore, closed-forms for all these moments can be computed, leading to computability of the closed-form of $\mathbb{E}(M)$.

Let \mathcal{P} be a normalized program, $x \in \text{Vars}(\mathcal{P})$, $k, n \in \mathbb{N}$ arbitrary, and \mathcal{M} be the set of all monomials over $\text{Vars}(\mathcal{P})$ with the powers of finite variables d bounded by the number of possible values of d (higher powers can be reduced as of Theorem 4).

Recurrences over Moments. Note that, given the syntax of probabilistic programs and properties of expectation, there is a natural way to express $\mathbb{E}(M_{n+1})$ as $\sum_{N \in \mathcal{M}^*} c_N \mathbb{E}(N_n)$ for any $M \in \mathcal{M}$, for some finite set $\mathcal{M}^* \subset \mathcal{M}$, and non-zero constants c_N . Intuitively, \mathcal{M}^* is the set of monomials that appear in the recurrence for $\mathbb{E}(M)$. We define the $*$ operator to give such a set for any monomial, and extend the definition to sets by

$$S^* = \bigcup_{M \in S} M^*.$$

The exact expression can be computed from $\mathbb{E}(M_{n+1})$ by replacing variables appearing in M by their assignments. Those are, in a normalized program, of the form

$$x = a_0 \{p_0\} \dots \{p_{i-1}\} a_i [C_x] d_x,$$

or

$$x = \text{Dist}[C_x] d_x$$

for some admissible distribution $Dist$, for which we rewrite $\mathbb{E}(M' \cdot x_{n+1}^k)$ to

$$\mathbb{E}\left(M' \left(d_x[\neg C_x] + \sum p_i a_i^k[C_x]\right)\right), \quad (5)$$

or to

$$\mathbb{E}(M' d_x[\neg C_x]) + \mathbb{E}(M'[C_x]) \mathbb{E}(Dist^k), \quad (6)$$

respectively, with the monomial M' not containing x . Variables in conditions $[C_x]$ are all finite, since they come from branch conditions. We further simplify the expressions of Equations (5)-(6) by replacing the logical conditions $[C_x]$ that evaluate to 1 whenever variables satisfy the condition $[C_x]$ and to 0 otherwise. It is possible to write any logical conditions over finitely valued variables as such polynomials ([Stankovič et al. 2022]), with $[x = c] := \prod_{d \in \omega(x) \setminus \{c\}} \frac{x-d}{c-d}$, $[\neg C] = 1 - [C]$, and $[C_1 \wedge C_2] = [C_1] \cdot [C_2]$, where $\omega(x)$ is the set of possible values of x . By converting conditions to polynomials in the equation above, we get a polynomial over moments of program variables. Replacing variables of iteration $n+1$ from last to first (by their appearance in \mathcal{P} 's loop body) we achieve an expression in the desired form.

Ordering. Now that we can compute the recurrences, we need to introduce the order for monomials, such that the monomials appearing in the recurrence for M are not larger than M . We will need this to show that computing the recurrences as described above is, indeed, a finite process. Intuitively, a variable y is larger than (or equal to) x if it depends on x . Mutually dependent variables will form an equivalence class. We then extend the order to monomials based on their degrees with respect to the variables' equivalence classes.

More formally, let \leq be a smallest total preorder on variables such that $x \leq y$ whenever y depends on x . We write $x < y$ iff $x \leq y$ and $y \not\leq x$, and $x \sim y$ iff $x \leq y$ and $y \leq x$. Let \tilde{x} be the equivalence class of x induced by \sim .

We extend \leq to a preorder on the set of monomials \mathcal{M} . For every monomial M and a variable x we consider the degree $\deg(\tilde{x}, M)$ of M in the equivalence class of \tilde{x} ¹. We associate M with the sequence of $\deg(\tilde{x}, M)$ for all variable equivalence classes, ordered with respect to \leq . Then $<$, \sim , and equivalence classes $(-)$ follow naturally from \leq .

By Theorem 4 and the definition of \leq , equivalence class \tilde{M} is finite for each $M \in \mathcal{M}$. Let \mathcal{M}_\sim be the set of equivalence classes of \sim . Preorder \leq induces a partial order \leq_\sim on \mathcal{M}_\sim . Since \leq is total and from properties of monomials, \leq_\sim is a well-order. We will write \leq instead of \leq_\sim when the meaning is clear from context.

From the restriction on polynomial dependence in \mathcal{P} , we further have $N \leq M$ for any $N \in M^*$ and $M \in \mathcal{M}$.

Finite $\mathcal{S}^{(Q)}$. We show, that for any monomial Q , there is a finite set $\mathcal{S}^{(Q)} \subset \mathcal{M}$ containing Q such that $M^* \subset \mathcal{S}^{(Q)}$ for any $M \in \mathcal{S}^{(Q)}$.

Let

$$\mathcal{S}^{(Q)} = \tilde{Q} \cup \bigcup_{\substack{A \in \tilde{Q}^* \\ A < Q}} \mathcal{S}^{(A)}. \quad (7)$$

Clearly $Q \in \mathcal{S}^{(Q)}$ and $M^* \subset \mathcal{S}^{(Q)}$ for any $M \in \mathcal{S}^{(Q)}$ by construction. We are left to show that $\mathcal{S}^{(Q)}$ is finite for all $Q \in \tilde{Q} \in \mathcal{M}_\sim$, which we can do by transfinite induction over \mathcal{M}_\sim .

For the base case we have to show that $\mathcal{S}^{(Q)}$ is finite for all Q in $\tilde{1}$. Since $\tilde{1} = \{\prod_{d \in F} d^{\lambda_d} \mid \lambda_d \leq m_d\}$, where F is the set of finite program variables and m_d are upper bounds on their powers as of

¹ $\deg(\tilde{x}, M) := \sum_{x \in \tilde{x}} \deg(x, M)$, where $\deg(x, M)$ is the degree of x in M .

Theorem 4, $\mathcal{S}^{(Q)} = \widetilde{1}$ is finite for all $Q \in \widetilde{1}$. Suppose $\mathcal{S}^{(A)}$ is finite for all $A < Q$. Since \widetilde{Q}^* is finite, so is the union in (7) and, as a result, $\mathcal{S}^{(Q)}$.

Moments. A system of linear recurrences with constant coefficients can be constructed for monomials in $\mathcal{S}^{(Q)}$. Therefore, the closed-form of any $E(M) \in \mathcal{M}$ exists and is computable. \square

We now turn back to Theorem 6 and establish its validity. The crux of our proof below relies on the fact that our transformations computing normal forms of \mathcal{P} (see Section 3.3) are \mathcal{P} -preserving.

PROOF (OF THEOREM 6). By Theorem 3 (Normal Form Termination), \mathcal{P} can be transformed to a normalized loop \mathcal{P}_N . By Lemma 7 (Non-Dependency Preservation), \mathcal{P}_N satisfies all conditions from Lemma 8 (Normal Moment-Computability). Thus, \mathcal{P}_N is moment-computable and the moments are equivalent to those of \mathcal{P} for $x \in \text{Vars}(\mathcal{P})$ by Theorem 3 (Normal Form Correctness). \square

The proofs of Theorem 6 and Lemma 8 are constructive and describe a procedure to compute (higher) moments of program variables by (1) transforming a probabilistic loop into a normal form according to Theorem 3, (2) constructing a system of linear recurrences as in the proof of Lemma 8 and (3) solving the system of linear recurrences with constant coefficients. In the following example, we illustrate the whole procedure on the running example from Figure 1.

Example 5. We return to the probabilistic loop \mathcal{P} from Figure 1. A normal form \mathcal{P}_N for \mathcal{P} was given in Figure 4. To compute a closed-form of the expected value of the program variable z , we will model $\mathbb{E}(z_n)$ as a system of linear recurrences according to the proof of Lemma 8. For a cleaner presentation we will refer to the variable toggle by t . Note that the t is binary. To construct the recurrence for $\mathbb{E}(z_n)$, we start with $\mathbb{E}(z_{n+1})$ at the assignment of z in \mathcal{P}_N and repeatedly replace variables by the right-hand side of their assignment starting from z 's assignment and stopping at the top of the loop body. Throughout the process we use the linearity of expectation to “push” \mathbb{E} inwards:

$$\begin{aligned}
& \mathbb{E}(z_{n+1}) \\
& \quad \downarrow \text{assignment of } z \\
& \mathbb{E}([t_{n+1} = 0](z_n + y_{n+1}) \{1/4\} [t_{n+1} = 0](z_n - y_{n+1})) + [t_{n+1} \neq 0]z_n \\
& \quad \downarrow \text{replace (in)equalities by polynomials} \\
& \mathbb{E}(((1 - t_{n+1})(z_n + y_{n+1}) \{1/4\} (1 - t_{n+1})(z_n - y_{n+1})) + t_{n+1}z_n) \\
& \quad \downarrow \mathbb{E} \text{ on categorical} \\
& \frac{1}{4}\mathbb{E}((1 - t_{n+1})(z_n + y_{n+1})) + \frac{3}{4}\mathbb{E}((1 - t_{n+1})(z_n - y_{n+1})) + \mathbb{E}(t_{n+1}z_n) \\
& \quad \downarrow \text{simplify; } \mathbb{E} \text{ linearity} \\
& \mathbb{E}(z_n) + \frac{1}{2}\mathbb{E}(t_{n+1}y_{n+1}) - \frac{1}{2}\mathbb{E}(y_{n+1}) \\
& \quad \downarrow \text{similarly, replace } y_{n+1}, x_{n+1}, t_{n+1} \\
& \mathbb{E}(z_n) - \frac{1}{6}\mathbb{E}(t_n x_n) - \frac{1}{2}\mathbb{E}(t_n y_n) - \frac{1}{6}\mathbb{E}(t_n x_n^2) + \frac{1}{12}\mathbb{E}(t_n) + \frac{1}{6}\mathbb{E}(t_n z_n).
\end{aligned}$$

The last line of the calculation represents the recurrence equation of the expected value of z . For every monomial in the recurrence equation of $\mathbb{E}(z_n)$ (that is: tx , ty , tx^2 , t , tz), we compute its respective recurrence equation and recursively repeat this procedure which eventually terminates according to Lemma 8. In the end we are faced with a system of linear recurrences of the expected values of the

monomials $z, tx, ty, x, y, tx^2, t, tz, x^2$ and 1 with recurrence matrix

$$\begin{bmatrix} 1 & -\frac{1}{6} & -\frac{1}{2} & 0 & 0 & -\frac{1}{6} & \frac{1}{12} & \frac{1}{6} & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & \frac{3}{2} & 0 & 0 & 0 \\ 0 & \frac{1}{3} & 0 & 0 & 1 & \frac{1}{3} & -\frac{1}{6} & -\frac{1}{3} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 & \frac{5}{2} & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

Using any computer algebra system we arrive at the closed-form solution for the expected value of z , $\mathbb{E}(z_n)$, parameterized by the loop counter n :

$$\begin{aligned} \mathbb{E}(z_n) = & \frac{883}{32} + \frac{29n}{16} - \frac{201}{20} 2^{-n} 6^{\frac{n}{2}} - \frac{67}{20} 2^{-n} 6^{\frac{1+n}{2}} - \frac{37}{10} 3^{-n} 6^{\frac{n}{2}} \\ & - \frac{37}{20} 3^{-n} 6^{\frac{1+n}{2}} - \frac{201}{20} 6^{\frac{n}{2}} \left(\frac{-1}{2}\right)^n - \frac{37}{10} 6^{\frac{n}{2}} \left(\frac{-1}{3}\right)^n + \frac{67}{20} 6^{\frac{1+n}{2}} \left(\frac{-1}{2}\right)^n \\ & + \frac{37}{20} 6^{\frac{1+n}{2}} \left(\frac{-1}{3}\right)^n + \frac{9}{16} n (-1)^n + \frac{29}{32} (-1)^n + \frac{9}{16} n^2. \end{aligned}$$

This example highlights the strength of algebraic techniques for probabilistic program analysis in comparison to constraint-based methods employing templates. We are not aware of any template-based method able to handle functions of the complexity of $\mathbb{E}(z_n)$. Our tool Polar is able to find the closed-form of $\mathbb{E}(z_n)$ in under one seconds (see Section 7).

5.1 Guarded Loops

When we model a guarded probabilistic loop **while** $\phi: \dots$ as an infinite loop **while** $\star: \text{if } \phi: \dots$, we impose the same restrictions on ϕ as on if-conditions (that means ϕ only contains finite variables) to guarantee computability and correctness. The k th moment of x after termination is then given by

$$\lim_{n \rightarrow \infty} \mathbb{E}(x_n^k \mid \neg \phi_n) = \lim_{n \rightarrow \infty} \frac{\mathbb{E}(x_n^k \cdot [\neg \phi_n])}{\mathbb{E}([\neg \phi_n])}. \quad (8)$$

If the limit exists, we can use standard methods from computer algebra to compute it, as the (higher) moments our approach computes are given as exponential polynomials [Gruntz 1996].

Example 6. Consider the following loop, in which x after termination is geometrically distributed with parameter $1/2$:

```
x, stop = 0, 0
while stop == 0:
    stop = Bernoulli(1/2)
    x = x + 1
end
```

With Equation 8 and the techniques from this section we get:

$$\mathbb{E}(x) = \lim_{n \rightarrow \infty} \mathbb{E}(x_n \mid \text{stop}_n = 1) = \lim_{n \rightarrow \infty} \frac{\mathbb{E}(x_n \cdot \text{stop}_n)}{\mathbb{E}(\text{stop}_n)} = \lim_{n \rightarrow \infty} \frac{-n2^{-n} - 2^{1-n} + 2}{1 - 2^{-n}} = 2.$$

Moreover, whenever the variables in the loop guard are not probabilistic, traditional techniques can be applied to determine the number of loop iterations n which can then be plugged into the (higher) moments computed by our work. Apart from the guarded loops, many systems show the

type of infinite behavior naturally modeled with infinite loops, such as probabilistic protocols or dynamical systems.

5.2 Infinite If-Conditions

Theorem 6 on moment-computability requires the variables in all if-conditions to be finite. Nevertheless, in some cases, if-conditions containing infinite variables can be handled by our approach. Let \mathcal{P} be a probabilistic loop containing an if-statement with condition F **and** I where F contains only finite variables and I contains infinite variables. Without loss of generality, no variable in I is assigned in or after the if-statement. Let the transformation removing I be defined by

$$\text{if } F \text{ and } I : \text{Branch} \quad \mapsto \quad \begin{array}{l} t = \text{Bernoulli}(p) \\ \text{if } F \text{ and } t == 1 : \\ \text{Branch} \text{ end} \end{array}$$

where t is a fresh variable and $p := \mathbb{P}(I)$ (potentially symbolic). Then, the transformation preserves the distributions of all $x \in \text{Vars}(\mathcal{P})$ under the following assumptions:

- (1) I is iteration independent, meaning for every variable x in I neither x nor any variable x depends on (as of Definition 15) has a self-dependency.
- (2) I is statistically independent from F and all conditions in Branch .
- (3) For every assignment A in Branch and every variable x in A which has been assigned before A , it holds that I and x are statistically independent.

Assumption 1 ensures that $\mathbb{P}(I) = \mathbb{E}([I])$ is constant. Assumption 2 and 3 further ensure that $\mathbb{E}([I])$ can always be “pulled out” (that means $\mathbb{E}([I]x) = \mathbb{E}([I])\mathbb{E}(x)$) in the construction of the recurrences. Assumptions 1-3 can often be checked automatically.

Example 7. Consider the statement **if** $g < 1/2$: $\text{sum} = \text{sum} + x$ of the program from Figure 1, where the value of g is drawn from a standard normal distribution. In this case, the transformation’s parameter p represents $\mathbb{P}(\text{Normal}(0, 1) < 1/2)$, but is left symbolic for the moment computation. The integral $\mathbb{P}(\text{Normal}(0, 1) < 1/2)$ can be solved separately and the result be substituted for p .

5.3 On the Necessity of the Conditions Ensuring Moment-Computability

Theorem 6 states two conditions that are sufficient to ensure that the closed-forms of the program variables’ higher moments always exist and are computable. Condition 1 enforces that there is no variable with potentially infinite values with a polynomial self-dependency. Condition 2 demands that all variables appearing in if-conditions are finite. We argue that both conditions are necessary, in the sense that if either of the conditions does not hold, the existence or computability of the variable moments’ closed-forms cannot be guaranteed.

Condition 1. Relaxing condition 1 of Theorem 6 means that we allow for polynomial self-dependencies of non-finite variables. The *logistic map* [May 1976] is a quadratic first-order recurrence defined by $x_{n+1} = r \cdot x_n(1 - x_n)$ and well-known for its chaotic behavior. A famous fact about the logistic map is that it does not have an analytical solution for most values of r [Maritz 2020]. By neglecting condition 1, we can easily devise a loop modeling the logistic map:

```
while ★:
  x = r · x (1 - x)
end
```

The value of the program variable x after iteration n is equal to the n th term of the logistic map. This means, for most values of r and initial values of x , there does not exist an analytical closed-form solution for the program variable x . Moreover, our counter-example illustrates that

condition 1 is necessary already for programs with a single variable and without stochasticity and if-statements.

Condition 2. Loosening condition 2 of Theorem 6 and allowing for non-finite variables in if-conditions renders our programming model Turing-complete. Intuitively, one can model a Turing machine's tape with two variables l and r such that the binary representation of l represents the tape's content left of the read-write-head. The binary representation of r represents the tape's content at the position of the read-write-head and towards the right. The least significant bit of r is the current symbol the Turing machine is reading. We can extract the least significant bit of r in our programming model (and neglecting condition 2) by introducing a variable lsb and using a single if-statement involving non-finite variables: whenever the loop changes the value of r , we set $lsb := r$. The while-loop's body is of the form “**if** $lsb > 1$: $lsb = lsb - 2$ **else** *transitions* **end**”. The Turing machine's transition table can be encoded using if-statements. Writing and shifting can be accommodated for by multiplying by 2 or $1/2$ and using addition and subtraction. The Turing machine's state can be modelled by a single finite variable. Therefore, by dropping condition 2, being able to model the program variables by linear recurrences would give rise to a decision procedure for the Halting problem: assume we introduce a variable *terminated* which is initialized to 0 before the loop and set to 1 whenever the Turing-machine terminates. If *terminated* can be modelled by a linear recurrence of order k , it suffices to check the first k values of the recurrence to determine whether or not *terminated* is always 0 [Kauers and Paule 2011] and the Turing-machine does not terminate. As the Halting problem is well-known to be undecidable, condition 2 is necessary to guarantee that the program variables can be modelled by linear recurrences, even without stochasticity and polynomial arithmetic.

6 USE-CASES OF HIGHER MOMENTS

For probabilistic loops, computing closed-forms of the variables' (higher) moments poses a technique for synthesizing quantitative invariants: Given a program variable x and a closed-form $f(n)$ of its k th moment, the equation $\mathbb{E}(x_n^k) = f(n)$ is an invariant. Moreover, closed-forms of raw moments can be converted into closed-forms of *central* moments, such as variance, skewness or kurtosis (cf. Section 2.1). In addition, this section provides hints on two further use-cases of higher moments of probabilistic loops: (i) deriving tail probabilities (Section 6.1) and (ii) inferring distributions of random variables from their moments (Section 6.2).

6.1 From Moments to Tail Probabilities

Tail probabilities measure the probability that a random variable surpasses some value. The mathematical literature contains several inequalities providing upper- or lower bounds on tail probabilities given (higher) moments [Boucheron et al. 2013]. Two examples are *Markov's inequality* for upper and the *Paley-Zygmund inequality* for lower bounds.

Theorem 9 (Markov's Inequality). *Let X be a non-negative random variable, and $t \geq 0$, then*

$$\mathbb{P}(X \geq t) \leq \frac{\mathbb{E}(X^k)}{t^k}.$$

Theorem 10 (Paley-Zygmund Inequality). *Let X be a random variable with $X \geq t$ almost-surely. Then*

$$\mathbb{P}(X > t) \geq \frac{(\mathbb{E}(X) - t)^2}{\mathbb{E}(X^2) - 2t\mathbb{E}(X) + t^2}.$$

Example 8. *For Herman's Self-Stabilization program from Figure 2 almost-surely $\text{tokens} \in \{0, 1, 2, 3\}$. With the techniques from previous sections, we can compute the first two moments $\mathbb{E}(\text{tokens}_n) =$*

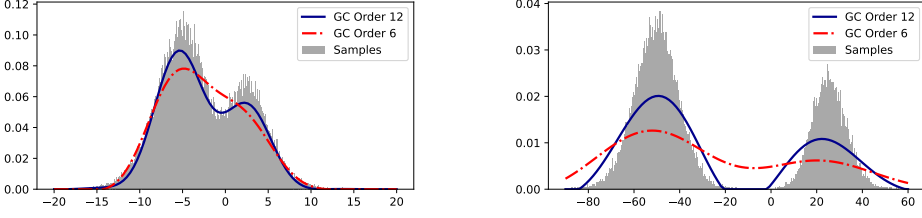


Fig. 5. The empirical density of program variable x for the benchmark *Bimodal* (cf. Table 1) and loop iterations 10 (left) and 100 (right) obtained by 10^5 samples, together with two approximations using the Gram-Charlier A Series with 6 (red dashed lines) and 12 (blue solid lines) moments.

$1 + 2 \cdot 4^{-n}$ and $\mathbb{E}(\text{tokens}_n^2) = 1 + 8 \cdot 4^{-n}$. Markov's inequality (Theorem 9) gives us the upper bound $\mathbb{P}(\text{tokens}_n \geq 2) \leq 1/2 + 4^{-n}$ using the first moment and $\mathbb{P}(\text{tokens}_n \geq 2) \leq 1/4 + 2 \cdot 4^{-n}$ utilizing the second moment.

For the Paley-Zygmund inequality (Theorem 10) both the first and the second moment are required, yielding the lower bound $\mathbb{P}(\text{tokens}_n \geq 2) = \mathbb{P}(\text{tokens}_n > 1) \geq 4^{-n}$. The theorem's precondition that almost-surely $\text{tokens} \geq 1$ might not be apparent at first sight. We take this for granted for now and will clarify this fact in Example 9.

6.2 From Moments to Distributions

For finite random variables, their full distribution can be recovered from finitely many moments. More precisely, given a random variable X with m possible values, the distribution of X can be recovered from its first $m-1$ moments, as the following theorem states:

Theorem 11. Let X be a random variable over $\{a_1, \dots, a_m\}$ and $p_i := P(X = a_i)$. The values p_i are the solutions of the system of linear equations given by $\sum_{i=1}^m p_i a_i^j = \mathbb{E}(x^j)$ for $0 \leq j < m$.

Example 9. Consider Herman's Self-Stabilization program from Figure 2. In Example 8 we obtained upper and lower bounds for tail probabilities of the *tokens* variable using the first one or two moments. With the first three moments we can fully recover the distribution of the *tokens* variable. We have that $\text{tokens} \in \{0, 1, 2, 3\}$. Let $p_i := \mathbb{P}(\text{tokens}_n = i)$ for $0 \leq i \leq 3$. By Theorem 11, we get the following system:

$$\begin{aligned} p_0 + p_1 + p_2 + p_3 &= \mathbb{E}(\text{tokens}_n^0) = 1, \\ p_1 + 2p_2 + 3p_3 &= \mathbb{E}(\text{tokens}_n) = 1 + 2 \cdot 4^{-n}, \\ p_1 + 4p_2 + 9p_3 &= \mathbb{E}(\text{tokens}_n^2) = 1 + 8 \cdot 4^{-n}, \\ p_1 + 8p_2 + 27p_3 &= \mathbb{E}(\text{tokens}_n^3) = 1 + 26 \cdot 4^{-n}. \end{aligned}$$

The solution can be obtained using standard techniques and tools, yielding $p_0 = 0$; $p_1 = 1 - 4^{-n}$; $p_2 = 0$; $p_3 = 4^{-n}$.

Note that probabilities are given as functions of the loop iteration n . Moreover, the solution shows that almost-surely $\text{tokens} \geq 1$, which we assumed to be true in Example 8.

The distributions of program variables with potentially infinitely many values, including continuous variables, cannot be, in general, fully reconstructed from finitely many moments. However, expansions such as the Gram-Charlier A Series [Kolassa 2013] can be used to approximate a probability density function using finitely many moments. Figure 5 illustrates how exact moments computed by our approach can be used to approximate unknown probability density functions of

program variables. While Figure 5 shows approximations for specific loop iterations, we emphasize that the symbolic nature of our approach allows for approximating the densities of program variables for all – potentially infinitely many – loop iterations simultaneously. Therefore, our technique is constant in the number of loop iterations, whereas sampling has linear complexity. We compute the approximations from Figure 5 for all infinitely many loop iterations in ~ 22 seconds, with the experimental setup from Section 7. In comparison, sampling the loop 10^5 times takes ~ 3.6 minutes for loop iteration 10 and ~ 33 minutes for loop iteration 100.

7 IMPLEMENTATION AND EVALUATION

Implementation. The program transformations (Section 3) and (higher) moment computation (Section 5) are implemented in the new tool Polar. For automatically inferring finiteness of program variables, we use a standard approach based on *abstract interpretation*. Polar is implemented in python3, consisting of ~ 3300 LoC, and uses the packages sympy² and symengine³ for symbolic manipulation of mathematical expressions. Together with all our benchmarks, Polar is publicly available at <https://github.com/probing-lab/polar>.

Experimental Setting and Evaluation. The evaluation of our work is split into three parts. First, we evaluate Polar on the ability of computing higher moments of for 15 probabilistic programs exhibiting different characteristics (Section 7.1). Second, we compare Polar to the exact tool Mora [Bartocci et al. 2020b] which computes so-called *moment-based invariants* for a subset of our programming model (Section 7.2). Third, we compare our tool to approximate methods estimating program variable moments by confidence intervals through sampling (Section 7.3). All experiments have been run on a machine with a 2.6 GHz Intel i7 (Gen 10) processor and 32 GB of RAM. Runtime measurements are averaged over 10 executions.

7.1 Experimental Results with Higher Moments

Table 1 shows the evaluation of Polar on the program from Figure 1 and 14 benchmarks which are either from the literature on probabilistic programming [Barthe et al. 2016; Batz et al. 2021; Chakarov and Sankaranarayanan 2014; Gretz et al. 2013; Kwiatkowska et al. 2012], differential privacy schemes [Warner 1965] (*Randomized-Response*), Dynamic Bayesian Networks (*DBN-Umbrella*, *DBN-Component-Health*) or well-known stochastic processes (*Las-Vegas-Search*, *Pi-Approximation*, *Bimodal*). The benchmarks *Retransmission-Protocol* and *Hawk-Dove-Symbolic* were further generalized from their original definition by replacing concrete numbers with symbolic constants. This makes these benchmarks only harder as solutions to the generalized versions are solutions for the concretizations. Table 1 illustrates that Polar can compute higher moments for various probabilistic programs exhibiting different features, like circular variable dependencies, if-statements, and symbolic constants with finite, infinite, continuous, and discrete state spaces. Moreover, the table shows that the number of program variables is *not* the primary factor for the complexity of computing moments. For instance, the benchmarks *50-Coin-Flips* and *Duelling-Cowboys* have 101 and 4 program variables respectively. Nevertheless, the runtimes for computing first moments for the two benchmarks only differ by 0.3s. The complexity of computing moments lies in the complexity of the resulting systems of recurrences which depend on the concrete features present in the benchmarks like specific variable dependencies, symbolic constants, or degrees of polynomials.

Benchmark	#V	C/If/S/INF/CONT	Moment	RT
Running-Example (Fig. 1)	7	✓/✓/✓/✓/✓	$\mathbb{E}(z)$	0.67
Herman-3	10	✓/✓/✗/✗/✗	$\mathbb{E}(\text{tokens}^3)$	0.59
Las-Vegas-Search	3	✗/✓/✗/✓/✗	$\mathbb{E}(\text{found}^{20})$	0.39
Pi-Approximation	4	✗/✓/✗/✓/✓	$\mathbb{E}(\text{count}^3)$	0.49
50-Coin-Flips	101	✗/✓/✗/✓/✗	$\mathbb{E}(\text{total})$	0.93
Gambler-Ruin-Momentum	4	✓/✗/✓/✓/✗	$\mathbb{E}(x^3)$	3.02
Hawk-Dove-Symbolic	5	✗/✓/✓/✓/✗	$\mathbb{E}(\text{p1bal}^4)$	2.17
Variable-Swap	4	✓/✗/✗/✓/✓	$\mathbb{E}(x^{30})$	2.63
Retransmission-Protocol	4	✗/✓/✓/✓/✗	$\mathbb{E}(\text{fail}^3)$	0.57
Randomized-Response	7	✗/✓/✓/✓/✗	$\mathbb{E}(\text{p1}^3)$	0.73
Duelling-Cowboys	4	✓/✓/✓/✗/✗	$\mathbb{E}(\text{ahit})$	1.23
Martingale-Bet	4	✗/✓/✓/✓/✗	$\mathbb{E}(\text{capital}^3)$	9.06
Bimodal	5	✗/✓/✗/✓/✗	$\mathbb{E}(x^{10})$	4.71
DBN-Umbrella	2	✗/✓/✓/✗/✗	$\mathbb{E}(\text{umbrella}^5)$	0.85
DBN-Component-Health	3	✗/✓/✗/✗/✗	$\mathbb{E}(\text{obs}^5)$	0.29

Table 1. Evaluation of Polar on 15 benchmarks. All times are in seconds. #V = number of variables in benchmark; C = benchmark contains circular dependencies; If = benchmark contains if-statements; S = benchmark contains symbolic constants; INF = benchmark’s states space is infinite; CONT = benchmark’s state space is continuous; Moment = Moment to compute; RT = Total runtime.

7.2 Experimental Comparison to Exact Methods

To the best of our knowledge, Mora is the only other tool capable of computing higher moments for variables of probabilistic loops without templates – as described in [Bartocci et al. 2019]. Mora operates on so-called *Prob-solvable loops* which form a strict subset of our program model (Section 3). Prob-solvable loops do not admit circular variable dependencies, if-statements, or state-dependent distribution parameters. We compare Polar against Mora on the Mora benchmarks taken from [Chakarov and Sankaranarayanan 2014; Chen et al. 2015; Katoen et al. 2010; Kura et al. 2019]. Details can be found in Table 2. The experiments illustrate that Polar can handle all programs and moments that Mora can. Mora, however, cannot compute any moment for any program in Table 1. On simple benchmarks, Polar is slightly slower than Mora due to the constant overhead of the program transformations and type inference. On complex benchmarks Polar provides a significant speedup compared to Mora. For instance, for the *STUTTERING_C* benchmark Polar computes the moment $\mathbb{E}(s^3)$ in about 7 seconds, whereas Mora needs about 2 minutes.

7.3 Experimental Comparison with Sampling

For a probabilistic loop with program variable x the moment $\mathbb{E}(x_n^k)$ can be approximated for fixed k and n by sampling x_n^k and calculating the sample average or confidence intervals. Table 3 compares Polar to computing confidence intervals by sampling for $k = 1$ and $n = 10$. The table shows that our tool is able to compute precise moments in a fraction of the time needed to sample programs to

²<https://www.sympy.org>

³<https://github.com/symengine>

Benchmark		Mora	Polar	Benchmark		Mora	Polar
COUPON	$\mathbb{E}(c)$	0.19	0.1	STUTTERING_A	$\mathbb{E}(s)$	0.19	0.12
	$\mathbb{E}(c^2)$	0.23	0.11 (0.06)		$\mathbb{E}(s^2)$	0.8	0.23 (0.07)
	$\mathbb{E}(c^3)$	0.23	0.11		$\mathbb{E}(s^3)$	2.19	0.74
COUPON4	$\mathbb{E}(c)$	0.75	0.12	STUTTERING_B	$\mathbb{E}(s)$	0.21	0.11
	$\mathbb{E}(c^2)$	0.77	0.14 (0.07)		$\mathbb{E}(s^2)$	0.69	0.2 (0.07)
	$\mathbb{E}(c^3)$	0.84	0.15		$\mathbb{E}(s^3)$	1.5	0.63
RANDOM_WALK_1D	$\mathbb{E}(x)$	0.03	0.07	STUTTERING_C	$\mathbb{E}(s)$	0.59	0.2
	$\mathbb{E}(x^2)$	0.07	0.09 (0.06)		$\mathbb{E}(s^2)$	8.65	1.61 (0.07)
	$\mathbb{E}(x^3)$	0.05	0.09		$\mathbb{E}(s^3)$	127.7	7.2
SUM_RND_SERIES	$\mathbb{E}(x)$	0.18	0.1	STUTTERING_D	$\mathbb{E}(s)$	0.53	0.23
	$\mathbb{E}(x^2)$	0.72	0.23 (0.06)		$\mathbb{E}(s^2)$	3.63	1.14 (0.07)
	$\mathbb{E}(x^3)$	1.97	0.53		$\mathbb{E}(s^3)$	21.74	3.85
PRODUCT_DEP_VAR	$\mathbb{E}(p)$	0.31	0.12	STUTTERING_P	$\mathbb{E}(s)$	0.19	0.12
	$\mathbb{E}(p^2)$	1.16	0.25 (0.06)		$\mathbb{E}(s^2)$	0.77	0.35 (0.07)
	$\mathbb{E}(p^3)$	3.4	0.71		$\mathbb{E}(s^3)$	2.24	0.88
RANDOM_WALK_2D	$\mathbb{E}(x)$	0.06	0.08	SQUARE	$\mathbb{E}(y)$	0.2	0.12
	$\mathbb{E}(x^2)$	0.16	0.1 (0.07)		$\mathbb{E}(y^2)$	0.67	0.25 (0.07)
	$\mathbb{E}(x^3)$	0.1	0.1		$\mathbb{E}(y^3)$	1.64	0.45
BINOMIAL(p)	$\mathbb{E}(x)$	0.08	0.09				
	$\mathbb{E}(x^2)$	0.26	0.14 (0.07)				
	$\mathbb{E}(x^3)$	0.58	0.25				

Table 2. Comparison of Polar to Mora. The runtimes are in seconds per tool, benchmark, and moment. For Polar the comparison contains in brackets the seconds spent on parsing, normalizing, and type inference.

Benchmark	Moment	CI 100 CI 1.000 CI 10.000	$T_{10.000}$	Polar
Running-Example (Fig. 1)	$\mathbb{E}(z_{10})$	(−51.2, 0.67) (−58.8, −40.2) (−46.6, −40.5)	723.6s	0.76s
Retransmission-Protocol	$\mathbb{E}(\text{fail}_{10})$	(0.05, 0.19) (0.07, 0.11) (0.09, 0.11)	188.4s	0.34s
Variable-Swap	$\mathbb{E}(y_{10})$	(4.61, 8.04) (5.16, 6.29) (5.24, 5.60)	340.7s	0.14s
Hawk-Dove-Symbolic	$\mathbb{E}(p1bal_{10})$	(7.50, 11.0) (8.94, 10.4) (9.76, 10.2)	51.1s	0.27s

Table 3. Comparison of Polar to approximation through sampling. Polar = the tools runtime to compute the precise moment; CI N = an approximated 0.95-CI-interval from N samples; $T_{10.000}$ = the runtime for CI 10.000. The symbolic constant p in *Retransmission-Protocol* is set to 0.9 and for *Hawk-Dove-Symbolic* we set $v = 4$ and $c = 8$.

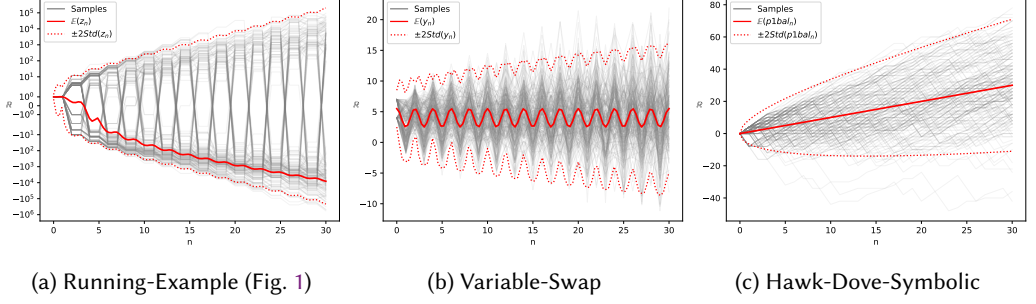


Fig. 6. Samples obtained by simulation plotted together with precise moments computed by Polar. In each benchmark, the thin gray lines are 200 samples over 30 iterations. The thick red line is the precise expected value. The dotted red lines are the expected values \pm twice the standard deviation given by the precise first two moments. Figure 6a is symmetric log scale.

achieve satisfactory confidence intervals. An advantage of sampling is that it is applicable for any probabilistic loop. However, by its nature, sampling fails to give any formal guarantees or hard bounds on the approximated moments. This is critical if the loop body contains branches that are executed with low probability. If applicable, Polar can provide *exact* moments for symbolic n (and involving other symbolic constants) faster than sampling can establish acceptable approximations. Moreover, even if the sampling of the loops is sped up by using a more efficient implementation, Polar enjoys complexity theoretical advantages. The complexity of sampling is linear in both the number of samples and the number of loop iterations. In contrast, Polar does not need to take multiple samples for higher precision as it symbolically computes the exact moments. Additionally, our method is *constant* in the number of loop iterations. With Polar, computing the moment for a specific loop iteration, say 10^5 , just amounts to evaluate the closed-form at 10^5 .

Figure 6 illustrates the importance of higher moments for probabilistic loops. The first moment provides a center of mass but contains no information on how the mass is distributed around this center. For this purpose higher moments are essential.

7.4 Evaluation Summary

Our experimental evaluation demonstrate that: (1) Polar can compute higher moments for a rich class of probabilistic loops with various characteristics, (2) Polar outperforms the state-of-the-art of moment computation for probabilistic loops in terms of supported programs and efficiency, and (3) Polar computes exact moments magnitudes faster than sampling can establish reasonable approximations.

8 RELATED WORK

Using recurrence equations to extract quantitative invariants of loops is a well-studied technique for non-probabilistic programs [Breck et al. 2020; de Oliveira et al. 2016; Farzan and Kincaid 2015; Humenberger et al. 2017, 2018; Kincaid et al. 2019, 2018; Kovács 2008; Rodríguez-carbonell and Kapur 2004].

A common approach to quantitatively and exactly analyze probabilistic programs is to employ probabilistic model checking techniques [Baier and Katoen 2008; Dehnert et al. 2017; Holtzen et al. 2021; Katoen et al. 2011; Kwiatkowska et al. 2011].

Exact inference for computing precise posterior distributions for probabilistic programs has been studied in [Claret et al. 2013; Gehr et al. 2016; Holtzen et al. 2020; Narayanan et al. 2016; Saad et al.

2021]. An interesting direction for future research is using our techniques to assist probabilistic inference in the presence of loops.

A different approach to characterize the distributions of program variables are statistical methods such as Monte Carlo and hypothesis testing [Younes and Simmons 2006]. Simulations are however performed on a chosen finite number of program steps and do not provide guarantees over a potentially infinite execution, such as unbounded loops, limiting thus their use (if at all) for invariant generation.

In [McIver and Morgan 2005], a deductive approach, the *weakest pre-expectation calculus*, for reasoning about PPs with discrete program variables is introduced. Based on the weakest pre-expectation calculus, [Katoen et al. 2010] presents the first template-based approach for generating linear quantitative invariants for PPs. Other works [Chen et al. 2015; Feng et al. 2017] also address the synthesis of non-linear invariants or employ *martingale* expressions [Barthe et al. 2016]. All of these works target a slightly different problem and, unlike our approach, rely on templates. The first data-driven technique for invariant generation for PPs is presented in [Bao et al. 2021].

Another line of related work comes with computing bounds over expected values [Bouissou et al. 2016; Chatterjee et al. 2020; Karp 1994] and higher moments [Kura et al. 2019; Wang et al. 2021]. The approach in [Bouissou et al. 2016] can provide bounds for higher moments and can handle non-linear terms at the price of producing more conservative bounds. In contrast, our approach natively supports probabilistic polynomial assignments and provides a precise symbolic expression for higher moments.

The technique presented in [Bartocci et al. 2019] automates the generation of so-called moment-based invariants for a subclass of PPs with polynomial probabilistic updates and sets the basis for fully automatic exact higher moment computation. Relative to our approach, [Bartocci et al. 2019] supports neither if-statements (thus also no guarded loops), state-dependent distribution parameters, nor circular variable dependencies. Our work establishes stronger theoretical foundations.

9 CONCLUSION

We describe a fully automated approach for inferring exact higher moments for program variables of a large class of probabilistic loops with complex control flow, polynomial assignments, symbolic constants, circular dependencies among variables, and potentially uncountable state spaces. Our work uses program transformations to normalize and simplify probabilistic programs while preserving their statistical properties. We propose a power reduction technique for finite program variables to ease the complex polynomial arithmetic of probabilistic programs. We prove soundness and completeness of our approach, by establishing the theory of moment-computable probabilistic loops. We demonstrate use cases of exact higher moments in the context of computing tail probabilities and recovering distributions from moments. Our experimental evaluation illustrates the applicability of our work, solving several examples whose automation so far was not yet supported by the state-of-the-art in probabilistic program analysis.

ACKNOWLEDGMENTS

This research was supported by the WWTF ICT19-018 grant ProbInG, the ERC Starting Grant SYMCAR 639270, ERC Consolidator Grant ARTIST 101002685, the Austrian FWF project W1255-N23, and the SecInt Doctoral College funded by TU Wien.

REFERENCES

- Christel Baier and Joost-Pieter Katoen. 2008. *Principles of Model Checking*. The MIT Press.
- Jialu Bao, Drashti Pathak, Justin Hsu, and Subhajit Roy. 2021. Data-Driven Invariant Learning for Probabilistic Programs. *ArXiv abs/2106.05421* (2021).
- Gilles Barthe, T. Espitau, L. M. F. Fioriti, and J. Hsu. 2016. Synthesizing Probabilistic Invariants via Doob’s Decomposition. In *CAV*.
- Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. 2012a. Probabilistic Relational Hoare Logics for Computer-Aided Security Proofs. In *MPC*.
- Gilles Barthe, Joost-Pieter Katoen, and Alexandra Silva. 2020. *Foundations of Probabilistic Programming*. Cambridge University Press.
- Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. 2012b. Probabilistic relational reasoning for differential privacy. In *POPL*.
- Ezio Bartocci, Laura Kovács, and Miroslav Stankovic. 2019. Automatic Generation of Moment-Based Invariants for Prob-Solvable Loops. In *ATVA*.
- Ezio Bartocci, Laura Kovács, and Miroslav Stankovic. 2020a. Analysis of Bayesian Networks via Prob-Solvable Loops. In *ICTAC*.
- Ezio Bartocci, Laura Kovács, and Miroslav Stankovic. 2020b. Mora - Automatic Generation of Moment-Based Invariants. In *TACAS*.
- Kevin Batz, Mingshuai Chen, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Philipp Schröder. 2021. Latticed k-Induction with an Application to Probabilistic Programs. In *CAV*.
- Stéphane Boucheron, Gábor Lugosi, and Pascal Massart. 2013. *Concentration Inequalities: A Nonasymptotic Theory of Independence*. OUP Oxford.
- Olivier Bouissou, Eric Goubault, Sylvie Putot, Aleksandar Chakarov, and Sriram Sankaranarayanan. 2016. Uncertainty Propagation Using Probabilistic Affine Forms and Concentration of Measure Inequalities. In *TACAS*.
- Jason Breck, John Cyphert, Zachary Kincaid, and T. Reps. 2020. Templates and recurrences: better together. *PLDI* (2020).
- Aleksandar Chakarov and Sriram Sankaranarayanan. 2014. Expectation Invariants for Probabilistic Program Loops as Fixed Points. (2014).
- Krishnendu Chatterjee, Hongfei Fu, Amir Kafshdar Goharshady, and Ehsan Kafshdar Goharshady. 2020. Polynomial invariant generation for non-deterministic recursive programs. In *PLDI*.
- Yu-Fang Chen, Chih-Duo Hong, Bow-Yaw Wang, and Lijun Zhang. 2015. Counterexample-Guided Polynomial Loop Invariant Generation by Lagrange Interpolation. In *CAV*.
- Yi Chou, Hansol Yoon, and Sriram Sankaranarayanan. 2020. Predictive Runtime Monitoring of Vehicle Models Using Bayesian Estimation and Reachability Analysis. In *IROS*.
- Guillaume Claret, Sriram K. Rajamani, Aditya V. Nori, Andrew D. Gordon, and Johannes Borgström. 2013. Bayesian Inference Using Data Flow Analysis. In *ESEC/FSE*.
- Steven de Oliveira, Saddek Bensalem, and Virgile Prevosto. 2016. Polynomial Invariants by Linear Algebra. In *ATVA*.
- Christian Dehnert, Sebastian Junges, Joost-Pieter Katoen, and Matthias Volk. 2017. A Storm is Coming: A Modern Probabilistic Model Checker. In *CAV*.
- Rick Durrett. 2019. *Probability: Theory and Examples*. Cambridge University Press.
- Azadeh Farzan and Zachary Kincaid. 2015. Compositional recurrence analysis. In *FMCAD*.
- Yijun Feng, Lijun Zhang, David N. Jansen, Naijun Zhan, and Bican Xia. 2017. Finding Polynomial Loop Invariants for Probabilistic Programs. In *ATVA*.
- Timon Gehr, Sasa Misailovic, and Martin T. Vechev. 2016. PSI: Exact Symbolic Inference for Probabilistic Programs. In *CAV*.
- Z. Ghahramani. 2015. Probabilistic machine learning and artificial intelligence. *Nature* (2015).
- F. Gretz, J.-P. Katoen, and A. McIver. 2013. Prinsys - On a Quest for Probabilistic Loop Invariants. In *QEST*.
- Dominik Gruntz. 1996. *On computing limits in a symbolic manipulation system*. Ph.D. Dissertation. ETH Zürich.
- W. Hastings. 1970. Monte Carlo Sampling Methods Using Markov Chains and Their Applications. *Biometrika* (1970).
- Ted Herman. 1990. Probabilistic Self-Stabilization. *Inf. Process. Lett.* (1990).
- Steven Holtzen, Sebastian Junges, Marcell Vazquez-Chanlatte, Todd D. Millstein, Sanjit A. Seshia, and Guy Van den Broeck. 2021. Model Checking Finite-Horizon Markov Chains with Probabilistic Inference. In *CAV*.
- Steven Holtzen, Guy Van den Broeck, and Todd Millstein. 2020. Scaling Exact Inference for Discrete Probabilistic Programs. *OOPSLA* (2020).
- Andreas Humenberger, Maximilian Jaroschek, and Laura Kovács. 2017. Automated Generation of Non-Linear Loop Invariants Utilizing Hypergeometric Sequences. *ISSAC* (2017).
- Andreas Humenberger, Maximilian Jaroschek, and Laura Kovács. 2018. Invariant Generation for Multi-Path Loops with Polynomial Assignments. In *VMCAI*.

- Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. 2019. On the hardness of analyzing probabilistic programs. *Acta Informatica* (2019).
- Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. 2016. Weakest Precondition Reasoning for Expected Run-Times of Probabilistic Programs. In *ESOP*.
- Richard M. Karp. 1994. Probabilistic Recurrence Relations. *J. ACM* 41, 6 (1994), 1136–1150.
- Joost-Pieter Katoen, Annabelle McIver, Larissa Meinicke, and Carroll C. Morgan. 2010. Linear-invariant generation for probabilistic programs: Automated support for proof-based methods. In *SAS*.
- Joost-Pieter Katoen, Ivan S. Zapreev, Ernst Moritz Hahn, Holger Hermanns, and David N. Jansen. 2011. The ins and outs of the probabilistic model checker MRMC. *Perform. Eval.* (2011).
- Manuel Kauers and Peter Paule. 2011. *The Concrete Tetrahedron - Symbolic Sums, Recurrence Equations, Generating Functions, Asymptotic Estimates*. Springer.
- Zachary Kincaid, Jason Breck, John Cyphert, and T. Reps. 2019. Closed forms for numerical loops. *POPL* (2019).
- Zachary Kincaid, John Cyphert, Jason Breck, and T. Reps. 2018. Non-linear reasoning for invariant synthesis. *POPL* (2018).
- J.E. Kolassa. 2013. *Series Approximation Methods in Statistics*. Springer New York.
- Laura Kovács. 2008. Reasoning Algebraically About P-Solvable Loops. In *TACAS*.
- Dexter Kozen. 1985. A Probabilistic PDL. *J. Comput. Syst. Sci.* (1985).
- Satoshi Kura, Natsuki Urabe, and Ichiro Hasuo. 2019. Tail Probabilities for Randomized Program Runtimes via Martingales for Higher Moments. In *TACAS*.
- Marta Z. Kwiatkowska, Gethin Norman, and David Parker. 2011. PRISM 4.0: Verification of Probabilistic Real-Time Systems. In *CAV*.
- Marta Z. Kwiatkowska, Gethin Norman, and David Parker. 2012. Probabilistic verification of Herman’s self-stabilisation algorithm. *Formal Aspects of Computing* (2012).
- Milton F. Maritz. 2020. A note on exact solutions of the logistic map. *Chaos: An Interdisciplinary Journal of Nonlinear Science* (2020).
- Robert M. May. 1976. Simple mathematical models with very complicated dynamics. *Nature* (1976).
- Annabelle McIver and Carroll Morgan. 2005. *Abstraction, Refinement and Proof for Probabilistic Systems*. Springer.
- Rajeev Motwani and Prabhakar Raghavan. 1995. *Randomized Algorithms*. Cambridge University Press.
- Praveena Narayanan, Jacques Carette, Wren Romano, Chung chieh Shan, and Robert Zinkov. 2016. Probabilistic Inference by Program Transformation in Hakaru (System Description). In *FLOPS*.
- Enric Rodríguez-carbonell and Deepak Kapur. 2004. Automatic Generation of Polynomial Loop Invariants: Algebraic Foundations. In *ISSAC*.
- Feras A. Saad, Martin C. Rinard, and Vikash K. Mansinghka. 2021. SPPL: probabilistic programming with fast exact symbolic inference. *PLDI* (2021).
- Konstantin Selyunin, Denise Ratasich, Ezio Bartocci, Md. Ariful Islam, Scott A. Smolka, and Radu Grosu. 2015. Neural Programming: Towards adaptive control in Cyber-Physical Systems. In *CDC*.
- Miroslav Stanković, Ezio Bartocci, and Laura Kovács. 2022. Moment-based analysis of Bayesian network properties. *Theoretical Computer Science* 903 (2022), 113–133.
- Di Wang, Jan Hoffmann, and Thomas Reps. 2021. Central Moment Analysis for Cost Accumulators in Probabilistic Programs. In *PLDI*.
- Stanley L. Warner. 1965. Randomized Response: A Survey Technique for Eliminating Evasive Answer Bias. *J. Am. Stat. Assoc.* (1965).
- Håkan L. S. Younes and Reid G. Simmons. 2006. Statistical probabilistic model checking with a focus on time-bounded properties. *Inf. Comput.* (2006).