Exact Bayesian Inference for Loopy Probabilistic Programs using Generating Functions

LUTZ KLINKENBERG[†], RWTH Aachen University, Germany
CHRISTIAN BLUMENTHAL, RWTH Aachen University, Germany
MINGSHUAI CHEN[†], Zhejiang University, China
DARION HAASE, RWTH Aachen University, Germany
JOOST-PIETER KATOEN, RWTH Aachen University, Germany

We present an exact Bayesian inference method for inferring posterior distributions encoded by probabilistic programs featuring possibly *unbounded loops*. Our method is built on a denotational semantics represented by *probability generating functions*, which resolves semantic intricacies induced by intertwining discrete probabilistic loops with *conditioning* (for encoding posterior observations). We implement our method in a tool called Prodigy; it augments existing computer algebra systems with the theory of generating functions for the (semi-)automatic inference and quantitative verification of conditioned probabilistic programs. Experimental results show that Prodigy can handle various infinite-state loopy programs and exhibits comparable performance to state-of-the-art exact inference tools over loop-free benchmarks.

CCS Concepts: • **Theory of computation → Program reasoning**; **Program semantics**; • **Mathematics of computing → Probabilistic inference problems**.

Additional Key Words and Phrases: probabilistic programs, quantitative verification, conditioning, Bayesian inference, denotational semantics, generating functions, non-termination

ACM Reference Format:
Lutz Klinkenberg, Christian Blumenthal, Mingshuai Chen, Darion Haase, and Joost-Pieter Katoen. 2024. Exact Bayesian Inference for Loopy Probabilistic Programs using Generating Functions. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 127 (April 2024), 31 pages. https://doi.org/10.1145/3649844

1 INTRODUCTION

Probabilistic programming is used to describe stochastic models in the form of executable computer programs. It enables fast and natural ways of designing statistical models without ever resorting to random variables in the mathematical sense. The so-obtained probabilistic programs [Barthe et al. 2020; Gordon et al. 2014; Holtzen et al. 2020; Kozen 1981; van de Meent et al. 2018] are typically normal-looking programs describing posterior probability distributions. They intrinsically code up randomized algorithms [Mitzenmacher and Upfal 2005] and are at the heart of approximate computing [Carbin et al. 2016] as well as probabilistic machine learning [van de Meent et al. 2018, Chapter 8]. One prominent example is Scenic [Fremont et al. 2023] – a domain-specific probabilistic

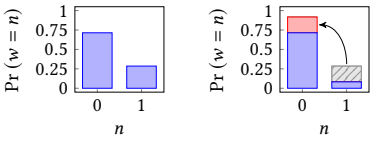
[†]The corresponding authors

Authors' addresses: Lutz Klinkenberg, lutz.klinkenberg@cs.rwth-aachen.de, RWTH Aachen University, Aachen, Germany; Christian Blumenthal, christian.blumenthal@rwth-aachen.de, RWTH Aachen University, Aachen, Germany; Mingshuai Chen, m.chen@zju.edu.cn, Zhejiang University, Hangzhou, China; Darion Haase, darion.haase@cs.rwth-aachen.de, RWTH Aachen University, Aachen, Germany; Joost-Pieter Katoen, katoen@cs.rwth-aachen.de, RWTH Aachen University, Aachen, Germany.

ACM 2475-1421/2024/4-ART127
https://doi.org/10.1145/3649844

```
{ w := 0 } [ 5/7 ] { w := 1 } ⨟
if ( w = 0 ) { c := poisson (6) }
else { c := poisson (2) } ⨟
observe ( c = 5 )
```

Prog. 1. The telephone operator.



(a) initial belief (b) updated belief

Fig. 1. The distribution of w in Prog. 1.

programming language to describe and generate scenarios for, e.g., robotic systems, that can be used to train convolutional neural networks; Scenic features the ability to declaratively impose (hard and soft) constraints over the generated models by means of *conditioning* via posterior observations. Moreover, a large volume of literature has been devoted to combining the strength of probabilistic and differentiable programming in a mutually beneficial manner; see [van de Meent et al. 2018, Chapter 8] for recent advancements in deep probabilistic programming.

Reasoning about probabilistic programs amounts to addressing various *quantities* like assertion-violation probabilities [Wang et al. 2021b], preexpectations [Batz et al. 2021; Feng et al. 2023; Hark et al. 2020], moments [Moosbrugger et al. 2022; Wang et al. 2021a], expected runtimes [Kaminski et al. 2018], and concentrations [Chakarov and Sankaranarayanan 2013; Chatterjee et al. 2016]. Probabilistic inference is one of the most important tasks in quantitative reasoning which aims to derive a program's posterior distribution. In contrast to sampling-based approximate inference, inferring the *exact* distribution has several benefits [Gehr et al. 2020], e.g., no loss of precision, natural support for symbolic parameters, and efficiency on models with certain structures.

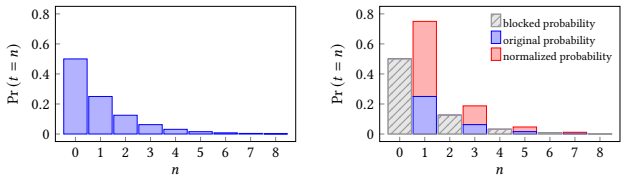
Exact probabilistic inference, however, is a notoriously difficult task [Ackerman et al. 2019; Cooper 1990; Kaminski et al. 2019; Olmedo et al. 2018; Roth 1996]; even for Bayesian networks, it is already PP-complete [Kwisthout 2009; Littman et al. 1998]. The challenges mainly arise from three program constructs: (i) unbounded `while`-loops and/or recursion, (ii) infinite-support distributions, and (iii) conditioning. Specifically, reasoning about probabilistic loops amounts to computing quantitative fixed points (see [Dahlqvist et al. 2020]) that are highly intractable in practice; admitting infinite-support distributions requires closed-form (i.e., finite) representations of program semantics; and conditioning "reshapes" the posterior distribution as per observed events thus yielding another layer of semantic intricacies (see [Ackerman et al. 2019; Bichsel et al. 2018; Olmedo et al. 2018]).

This paper proposes to use *probability generating functions* (PGFs) – a subclass of *generating functions* (GFs) [Wilf 2005] – to do exact inference for *discrete*, *loopy*, *infinite-state* probabilistic programs *with conditioning*, thus addressing challenges (i), (ii), and (iii), whilst aiming to push the limits of automation as far as possible by leveraging the strength of existing computer algebra systems like SymPy [Meurer et al. 2017] and GiNaC [Bauer et al. 2002; Vollinga 2006]. We extend the PGF-based semantics by Klinkenberg et al. [2020], which enables exact quantitative reasoning for, e.g., deciding probabilistic equivalence [Chen et al. 2022a] and proving non-almost-sure termination [Klinkenberg et al. 2020] for certain programs *without conditioning*. Orthogonally, Zaiser et al. [2023] recently employed PGFs to conduct exact Bayesian inference for conditioned probabilistic programs with infinite-support distributions yet *no loops*. Note that *having loops and conditioning intertwined* incurs semantic intricacies; see [Bichsel et al. 2018; Olmedo et al. 2018]. Let us illustrate our inference method and how it addresses such semantic intricacies by means of a number of examples of increasing complexity.

Conditioning in loop-free programs. Consider the loop-free program Prog. 1 producing an *infinite-support* distribution. It describes a telephone operator who is unaware of whether today is a weekday or weekend. The operator's initial belief is that with probability $5/7$ it is a weekday ($w = 0$) and thus

```
h := 1 ⨾
while ( h = 1 ) {
    { t := t + 1 } [ 1/2 ] { h := 0 }
} ⨾
observe ( t ≡ 1 (mod 2) )
```

Prog. 2. The odd geometric distribution.



(a) before conditioning (b) after conditioning

Fig. 2. Snippets of the distribution of t in Prog. 2.

with probability $2/7$ weekend ($w = 1$); see Fig. 0a. Usually, on weekdays there are 6 incoming calls per hour on average; on weekends this rate decreases to 2 calls – both rates are subject to a Poisson distribution. The operator observes 5 calls in the last hour, and the inference task is to compute the posterior distribution in which the initial belief is updated based on the observation. Our approach can automatically infer the updated belief (see Fig. 0b) with $\Pr(w = 0) = \frac{1215}{1215+2\cdot e^4} \approx 0.9175$. (Detailed calculations of the PGF semantics for Prog. 1 are given in Example 8 on page 11.)

Conditioning outside loops. Prog. 2 describes an iterative algorithm that repeatedly flips a fair coin – while counting the number of trials (t) – until seeing tails ($h = 0$), and observes that this number is odd. In fact, the `while`-loop produces a geometric distribution in t (cf. Fig. 1a), after which the `observe` statement "blocks" all program runs where t is even and normalizes the probabilities of the remaining runs (cf. Fig. 1b). Note that Prog. 2 features an unbounded looping behavior (inducing an infinite-support distribution) whose exact output distribution thus cannot be inferred by state-of-the-art inference engines, e.g., neither by (λ)PSI [Gehr et al. 2016, 2020], nor by the PGF-based approach in [Zaiser et al. 2023]. However, given a suitable loop invariant, our tool is able to derive the posterior distribution of Prog. 2 in an automated fashion: for any input with $t = 0$, the posterior is represented as the closed-form PGF

$$\frac{3\cdot T}{4-T^2} \;=\; \sum_{n=0}^{\infty} \underbrace{-3\cdot 2^{-2-n}\cdot\left(-1+(-1)^n\right)}_{\Pr(t=n\,\wedge\, h=0)} \cdot T^n H^0\,,$$

where T, H are formal *indeterminates* corresponding to the program variables t and h, respectively. From this closed-form PGF, we can extract various quantitative properties of interest, e.g., the expected value of t is $\mathbb{E}[t] = \left(\frac{\partial}{\partial T}\frac{3\cdot T}{4-T^2}\right)[H/0, T/1] = \frac{5}{3}$, or compute concentration bounds (aka tail bounds) such as $\Pr(t > 100) \leq \frac{5}{3\cdot 100} = \frac{1}{60}$ à la Markov's inequality [Dubhashi and Panconesi 2009].

Conditioning inside i.i.d. loops. As argued by Olmedo et al. [2018] and Bichsel et al. [2018], having loops and conditioning intertwined incurs semantic intricacies: Consider Prog. 3 – a variant of Prog. 2 where instead we observe $h = 1$ *inside* the `while`-loop. Prog. 3 features an *i.i.d. loop*, i.e., the set of states reached upon the end of different loop iterations are *independent and identically distributed*. This program is interesting since it conditions to a *zero-probability event*, i.e., the probability of infinitely often ignoring $h := 0$ is zero, which is important yet non-trivial to detect in general. Assigning a meaningful semantics to Prog. 3 is delicate: Intuitively, the `observe` statement prevents the `while`-loop from terminating since we always observe that we have taken the left branch, and therefore never set the termination flag $h = 0$. As a consequence, all runs that eventually would terminate are invalid as they violate the observation criterion. The single run

```
h := 1 ⨾
while ( h = 1 ) {
    { t := t + 1 } [ 1/2 ] { h := 0 } ⨾
    observe ( h = 1 )
}
```

Prog. 3. `observe` inside loop.

that does satisfy the criterion in turn is never able to exit the loop (cf. Section 3.4). In previous work on using PGFs (without conditioning) [Chen et al. 2022a; Klinkenberg et al. 2020], the semantics of non-termination is represented as subprobability distributions where the "missing" probability mass captures the probability of divergence. Zaiser et al. [2023] circumvent such semantic intricacies by syntactically imposing certainly terminating programs (due to the absence of loops and recursion). In our work, we distinguish non-termination behaviors from `observe` violations [Bichsel et al. 2018; Olmedo et al. 2018], which allows us to show that the `while`-loop in Prog. 3 is in fact equivalent to

$$\texttt{if}\ (h = 1)\ \{\texttt{observe}\ (\texttt{false})\}\ \texttt{else}\ \{\texttt{skip}\}$$

which in turn reduces to $\texttt{observe}\ (h \neq 1)$.

For certain programs, conditioning inside loops can be treated differently. These approaches include (1) *hoisting* [Olmedo et al. 2018] that removes observations completely from conditioned probabilistic programs, which however relies on intractable fixed point computations to hoist `observe` statements inside loops; (2) the *pre-image transformation* [Nori et al. 2014] that propagates observations backward through the program, which however cannot hoist the `observe` statement through probabilistic choices, as in Prog. 3; (3) the *ad hoc solution* that simply pulls the `observe` statement outside the loop, which however works only for special i.i.d. loops like Prog. 3: The `observe` statement in Prog. 3 can be equivalently moved downward to the outside of the loop, but such transformation does not generalize to non-i.i.d. loops (which may have data flow across different loop iterations) as exemplified below.

Conditioning inside non-i.i.d. loops. The probabilistic loop in Prog. 4 models a discrete sampler which keeps tossing two fair coins (h_1 and h_2) until they both turn tails. The `observe` statement in this program conditions to the event that at least one of the coins yields the same outcome as in the previous iteration, thereby imposing the global effect to "reset" the counter n and restart the program upon observation violations. This way of conditioning – that induces data dependencies across consecutive loop iterations – renders the loop non-i.i.d. and, as a consequence, no known tactic can be employed to pull the observation outside the loop. However, given a suitable invariant – in the form of a conditioned *loop-free* program that is equivalent to the loop – our method automatically infers that the posterior distribution is $-\frac{7\cdot N^2}{N^2+8\cdot N-16}$, where N is the formal indeterminate of the counter n (note that $h_1 = h_2 = h_1' = h_2' = 0$ on termination). Furthermore, our inference framework admits *parameters* in both programs and invariants for, e.g., encoding distributions with unknown probabilities like $\texttt{bernoulli}\ (p)$ with $p \in (0,1)$; it is capable of determining possible valuations of these parameters such that the given invariant is equivalent to the loop in question. The support of parameters in our approach enables template-based invariant synthesis (see, e.g., [Batz et al. 2023]) and model repair (cf. [Češka et al. 2019]), as detailed in Section 5.

$$
\begin{array}{l}
n \coloneqq 0\ ;\\
h_1 \coloneqq 1\ ;\ h_2 \coloneqq 1\ ;\ h_1' \coloneqq 1\ ;\ h_2' \coloneqq 1\ ;\\
\texttt{while}\ (\neg\,(h_1 = 0 \wedge h_2 = 0))\ \{\\
\quad h_1 \coloneqq \texttt{bernoulli}\ (1/2)\ ;\\
\quad h_2 \coloneqq \texttt{bernoulli}\ (1/2)\ ;\\
\quad \texttt{observe}\ \left(h_1 = h_1' \vee h_2 = h_2'\right)\ ;\\
\quad h_1' \coloneqq h_1\ ;\\
\quad h_2' \coloneqq h_2\ ;\\
\quad n \coloneqq n + 1\\
\}
\end{array}
$$

Prog. 4. The non-i.i.d. discrete sampler.

Approach. Fig. 3 sketches an overview of our inference approach: Given a prior distribution G and a loopy probabilistic program P with conditioning (at any place), our primary goal is to infer the posterior (sub-)distribution $[\![P]\!](G)$ as depicted by the upper row. Here, we interpret P as a distribution transformer $[\![P]\!](\cdot)$ that transforms G into $[\![P]\!](G)$, both represented as PGFs to encode possibly infinite-support distributions. To deal with the unbounded `while`-loop in P, we provide an invariant $I(p)$ in the form of a loop-free program parametrized by $p \in \mathbb{R}$, and aim to

synthesize parameter values under which $I(p)$ is semantically equivalent to P, i.e., they transform every possible prior distribution into the same posterior (sub-)distribution. We show that checking the program equivalence $[\![I(p)]\!] = [\![P]\!]$ (together with parameter synthesis) is decidable – via an extended technique of second-order PGFs – when $I(p)$ and P are restricted to a syntactic class of programs called `cReDiP` preserving closed-form PGFs. Once the equivalence is concluded, we can simply push the prior distribution G through the *loop-free* program $I(p)$ – as illustrated by the downward path in Figure 3 – and obtain the posterior (sub-)distribution $[\![P]\!](G)$, from which various quantitative queries can be addressed. To tackle conditioning, a key technical ingredient in our approach is to extend PGFs with an extra term X_4 keeping track of observation violations, which will eventually be normalized off to achieve the final, normalized (sub-)distribution.

Contributions. The main results of this paper are as follows.

- We present a PGF-based denotational semantics for discrete probabilistic `while`-programs where conditioning can occur at any place in the program. The basic technical ingredient is to extend PGFs with an extra term encoding the probability of violating observations as proposed by [Bichsel et al. 2018]. The semantics can treat conditioning in the presence of possibly diverging loops and captures conditioning on zero-probability events.
- This semantics extends the PGF-based semantics of [Chen et al. 2022a; Klinkenberg et al. 2020] for unconditioned programs and is shown to coincide with the Markov chain semantics in [Olmedo et al. 2018]. These correspondences indicate the adequacy of our semantics.
- Our PGF-based semantics readily enables exact inference for loop-free programs. We identify a syntactic class of almost-surely terminating programs for which exact inference for a

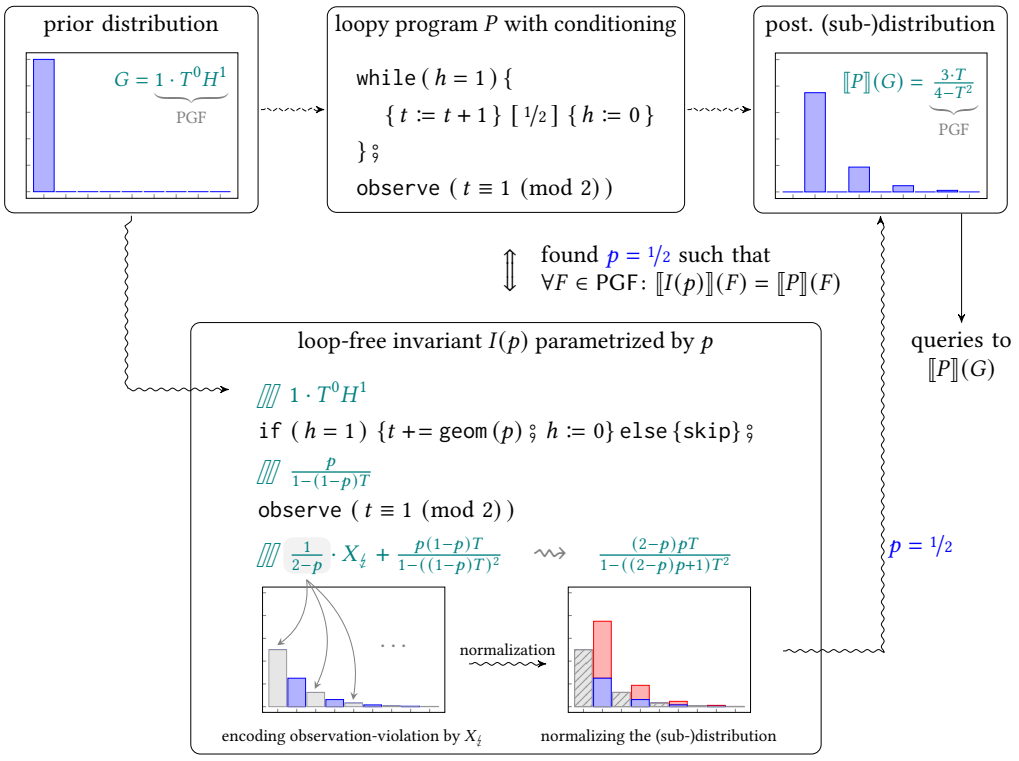


Fig. 3. A bird's-eye perspective of our approach.

`while`-loop coincides with inference for a straight-line program. Technically this is based on proving program equivalence.
- We show that, for this class of programs, our approach can be generalized towards parameter synthesis: Are a `while`-loop and a loop-free program that (both may) contain some parametric probability terms equivalent for some values of these unknown probabilities?
- We implement our method in a tool called Prodigy; it augments existing computer algebra systems with GFs for (semi-)automatic inference and quantitative verification of conditioned probabilistic programs. We show that Prodigy can handle many infinite-state loopy programs and exhibits comparable performance to state-of-the-art exact inference tools over benchmarks of loop-free programs.

Paper structure. Section 2 presents preliminaries on generating functions. Section 3 presents our extended PGF-based denotational semantics that allows for exact quantitative reasoning about probabilistic programs with conditioning. We dedicate Section 4 to the exact Bayesian inference for conditioned programs with loops leveraging the notions of invariants and equivalence checking. In Section 5, we identify the class of parametrized programs and invariants for which the problem of parameter synthesis is shown decidable. We report the empirical evaluation of Prodigy in Section 6 and discuss the limitations of our approach in Section 7. An extensive review of related work in probabilistic inference is given in Section 8. The paper is concluded in Section 9. Additional background materials, elaborated proofs, and details on the examples can be found in the appendix of the extended version [Klinkenberg et al. 2024a].

2 PRELIMINARIES ON GENERATING FUNCTIONS

Generating functions (GFs) constitute a versatile mathematical tool with extensive applications across various fields of mathematics and beyond [Wilf 2005]. They provide a systematic and elegant means of representing and manipulating sequences of numbers, rendering them essential for solving a diverse spectrum of mathematical problems in, e.g., enumerative combinatorics [Flajolet and Sedgewick 2009] and (discrete) probability theory [Johnson et al. 2005].

Formal power series. Generating functions, at their core, are *formal power series* (FPSs), which encode essential information about possibly infinite sequences of numerical values (of any type). The underlying principle is to represent the sequence as terms within an FPS (amenable to algebraic operations). Generating functions are classified as uni- or multivariate based upon the number of indeterminates. A *univariate generating function* in *formal indeterminate* X takes the form

$$F \;=\; \sum\nolimits_{n\in\mathbb{N}} a_n X^n \tag{1}$$

where a_n is the n-th number within the sequence. The "monomials" X^n are merely *position-holders* for the coefficients a_n and do not have any particular meaning. However, à la Klinkenberg et al. [2020] and Zaiser and Ong [2023], we interpret the indeterminate X with the corresponding program variable x and the exponent n with values of x; in this case, a_n is the probability of $x = n$.

Example 1 (Geometric Distribution as an FPS). Consider a discrete random (program) variable t which is geometrically distributed over $\mathbb{N}$ with parameter $1/2$. The probability mass function of t is given by $P_t(t = n) = 1/2^{n+1}$. We tabulate P_t using a sequence $(a_n)_{n\in\mathbb{N}} = (P_t(t = n))_n = 1/2, 1/4, 1/8, \ldots$. Encoding this sequence as a generating function in terms of FPSs via formal indeterminate T yields

$$\frac{1}{2} + \frac{1}{4}T + \frac{1}{8}T^2 + \frac{1}{16}T^3 + \frac{1}{32}T^4 + \frac{1}{64}T^5 + \frac{1}{128}T^6 + \frac{1}{256}T^7 + \cdots \tag{2}$$

where we uniquely associate terms of the power series to values of the sequence, e.g., the term $\frac{1}{8}T^2$ encodes the information that the probability of $t = 2$ is $1/8$.

Table 1. GF cheat sheet. f, g and X, Y are arbitrary GFs and indeterminates, resp. [Chen et al. 2022a].

Operation	Effect	Example
$f^{-1} = 1/f$	multiplicative inverse of f (if it exists)	$\frac{1}{1-XY} = 1 + XY + X^2Y^2 + \cdots$ because $(1 - XY)(1 + XY + X^2Y^2 + \cdots) = 1$
$f \cdot X$	shift in dimension X	$\frac{X}{1-XY} = X + X^2Y + X^3Y^2 + \cdots$
$f[X/0]$	drop terms containing X	$\frac{1}{1-0Y} = 1$
$f[X/1]$	projection[1] on Y	$\frac{1}{1-1Y} = 1 + Y + Y^2 + \cdots$
$f \cdot g$	discrete convolution (or Cauchy product)	$\frac{1}{(1-XY)^2} = 1 + 2XY + 3X^2Y^2 + \cdots$
$\partial_X f$	formal derivative in X	$\partial_X \frac{1}{1-XY} = \frac{Y}{(1-XY)^2} = Y + 2XY^2 + 3X^2Y^3 + \cdots$
$f + g$	coefficient-wise sum	$\frac{1}{1-XY} + \frac{1}{(1-XY)^2} = \frac{2-XY}{(1-XY)^2} = 2 + 3XY + 4X^2Y^2 + \cdots$
$a \cdot f$	coefficient-wise scaling (by scalar a)	$\frac{7}{(1-XY)^2} = 7 + 14XY + 21X^2Y^2 + \cdots$

In order to deal with multiple program variables $x_1, \ldots, x_k$, the form in Eq. (1) is generalized to a *multivariate* generating function of dimension $k \in \mathbb{N}$ as $F = \sum_{\mathbf{n} \in \mathbb{N}^k} a_{\mathbf{n}} \mathbf{X}^{\mathbf{n}}$, where $\mathbf{X} = (X_1, X_2, \ldots, X_k)$ is a vector of indeterminates and $\mathbf{X}^{\mathbf{n}}$ is the monomial $X_1^{n_1} X_2^{n_2} \cdots X_k^{n_k}$. Here, the term $a_{\mathbf{n}} \mathbf{X}^{\mathbf{n}}$ encodes that $(x_1, x_2, \ldots, x_k) = (n_1, n_2, \ldots, n_k)$ with probability $a_{\mathbf{n}}$. A k-dimensional GF F is called a *probability generating function* (PGF) if $\sum_{\mathbf{n} \in \mathbb{N}^k} a_{\mathbf{n}} \leq 1$ and $a_{\mathbf{n}} \geq 0$ for all $\mathbf{n} \in \mathbb{N}^k$ (cf. Eq. (2)). A PGF with $\sum_{\mathbf{n} \in \mathbb{N}^k} a_{\mathbf{n}} < 1$ represents a subprobability distribution and is called a *sub-PGF*.

Closed forms. The encoding as in Example 1 enables us to compress the infinite power series into a *closed form* using Taylor's theorem, that is, a finitely-represented function whose Taylor series developed at zero coincides with the GF. For instance, the closed form of Eq. (2) is given by $T \mapsto 1/(2-T)$ for all $|T| < 2$, as the Taylor series of $1/(2-T)$ is precisely $\frac{1}{2} + \frac{1}{4}T + \frac{1}{8}T^2 + \cdots$. Many important operations on infinite sequences of numbers – and their corresponding GF series – can be simulated by manipulating the closed-form expression instead. Using algebraic operations, this allows for computing, e.g., expected values, variances, higher-order moments, point probabilities, and tail bounds. For instance, the formal derivative $\frac{\mathrm{d}}{\mathrm{d}T} \frac{1}{2-T} = \frac{1}{(2-T)^2}$ evaluated at $T = 1$ yields the expected value $\mathbb{E}(t) = \frac{1}{(2-1)^2} = 1$. Table 1 summarizes some basic operations on GFs and their corresponding effects on the infinite sequences.

To effectively manipulate closed forms, we embed them in an algebraic structure – the (commutative) *ring of FPSs* $(\mathbb{R}[[\mathbf{X}]], +, \cdot, 0, 1)$. Here, $\mathbb{R}[[\mathbf{X}]]$ is the set of FPSs (of fixed dimension k):

$$F = \sum_{\mathbf{n} \in \mathbb{N}^k} [\mathbf{n}]_F \mathbf{X}^{\mathbf{n}}$$

with $[\cdot]_F \colon \mathbb{N}^k \to \mathbb{R}$, "+" (addition) and "$\cdot$" (multiplication) are binary operations defined as

$$F + G \triangleq \sum_{\mathbf{n} \in \mathbb{N}^k} ([\mathbf{n}]_F + [\mathbf{n}]_G)\, \mathbf{X}^{\mathbf{n}} \quad \text{and} \quad F \cdot G \triangleq \sum_{\mathbf{n}_1, \mathbf{n}_2 \in \mathbb{N}^k} ([\mathbf{n}_1]_F \cdot [\mathbf{n}_2]_G)\, \mathbf{X}^{\mathbf{n}_1 + \mathbf{n}_2},$$

and $0, 1 \in \mathbb{R}[[\mathbf{X}]]$ are neutral elements w.r.t. addition and multiplication, respectively. The multiplication $F \cdot G$ is in fact the discrete convolution of the two sequences F and G (aka, the *Cauchy product* of power series). Note that $F \cdot G$ is always well-defined because for all $\mathbf{n} \in \mathbb{N}^k$ there are *finitely* many $\mathbf{n}_1 + \mathbf{n}_2 = \mathbf{n}$ in $\mathbb{N}^k$. Moreover, every $F \in \mathbb{R}[[\mathbf{X}]]$ has an *additive inverse* $-F \in \mathbb{R}[[\mathbf{X}]]$ yet *multiplicative inverses* $F^{-1} = 1/F$ need not always exist.

Remark. Treating the closed form as a *function*, say $T \mapsto \frac{1}{2-T}$, and computing its Taylor series imposes – for the sake of well-definedness – the radius of convergence of the resulting series, i.e., $|T| < 2$. However, due to the underlying algebraic structure, we can safely write $\frac{1}{2-T} = \sum_{n\in\mathbb{N}} \frac{1}{2^{n+1}} T^n$ regardless of the fact whether $|T| < 2$: the sequences $2 - 1T + 0T^2 + \cdots$ and $\frac{1}{2} + \frac{1}{4}T + \frac{1}{8}T^2 + \cdots$ are *multiplicative inverse elements* to each other in $\mathbb{R}[[T]]$, i.e., their product is 1. We refer interested readers to [Chen et al. 2022b, Appendix D] for more details on convergence-related issues. ◁

In this paper, we are mainly concerned with *rational closed forms*, i.e., FPSs of the form $F = GH^{-1} = G/H$ where G, H are *polynomials* in $\mathbb{R}[[\mathbf{X}]]$ (i.e., G, H have finitely many non-zero coefficients).

3 GENERATING FUNCTION SEMANTICS

3.1 Semantics without Conditioning

Given a fixed input, the semantics of a probabilistic program is captured by its (posterior) probability distribution over the final (terminating) program states. In [Klinkenberg et al. 2020], the domain of discrete distributions is represented in terms of PGFs – elements from $\mathbb{R}[[\mathbf{X}]]$ – and a (conditioning-free) program is interpreted denotationally as a *distribution transformer* à la Kozen [Kozen 1981]. We recap this semantics for programs without conditioning by means of an example:

Example 2 (PGF Semantics without Conditioning). Consider Prog. 5 with input $G = 1 \cdot X^0C^0$, representing the joint prior distribution $\Pr(x = 0 \wedge c = 0) = 1$. The denotational PGF semantics of this program is computed in a *forward* manner per the annotation style in [Kaminski 2019]. Below, we show step-by-step how the prior distribution G is transformed into the joint posterior distribution $G' = 1/3 \cdot X^6C^5 + 2/3 \cdot C^3$, indicating that $\Pr(x = 6 \wedge c = 5) = 1/3$ and $\Pr(x = 0 \wedge c = 3) = 2/3$. We start by interpreting the first instruction of the program, i.e., the assignment of 1 to variable x, which results in the intermediate distribution $1 \cdot X^1$. Then, we descend into the left and right branches of the probabilistic choice statement. For the left branch, we interpret the semantics of $c := c + 5$ by multiplying the previous distribution $1 \cdot X^1C^0$ with C^5 which encodes the effect of increasing c by 5. The right branch is handled analogously by setting c to 3; this is done by first marginalizing the distribution $1 \cdot X^1C^0$ by substituting 1 for C and then multiplying the result with C^3. Now, we can combine the semantics for the two branches (left: XC^5; right: XC^3) via a weighted sum $1/3 \cdot XC^5 + 2/3 \cdot XC^3$ to represent the distribution after executing the probabilistic choice. Subsequently, we evaluate the conditional branching by recursively descending into the satisfying branch ($x := x + c$) and non-satisfying branch ($x--$) with their respective filtered inputs ($c > 4$: $1/3 \cdot XC^5$; $c \leq 4$: $2/3 \cdot XC^3$). Finally, we combine the two sub-results of the conditional branches and thus obtain $1/3 \cdot X^6C^5 + 2/3 \cdot C^3$. See [Klinkenberg et al. 2020] for semantics of more program constructs. ◁

```
/// 1                                  (= 1 · X^0 C^0)
x := 1
/// X
{ c := c + 5 } [ 1/3 ] { c := 3 }
/// 1/3 · XC^5 + 2/3 · XC^3
if ( c > 4 ) {
    /// 1/3 · XC^5
    x := x + c
    /// 1/3 · X^6 C^5
} else {
    /// 2/3 · XC^3
    x--
    /// 2/3 · C^3
}
/// 1/3 · X^6 C^5 + 2/3 · C^3
```

Prog. 5. PGF semantics for an observation-free program.

The representation of a posterior distribution in terms of a generating function comes with several benefits: (1) it naturally encodes common, *infinite-support* distributions like the geometric or Poisson distribution in compact, *closed-form* representations; (2) it allows for compositional

[1] Projection is not always well-defined, e.g., $\frac{1}{1-X+Y}[X/1] = \frac{1}{Y}$ is ill-defined, as Y is not invertible. It is, however, well-defined whenever used in this paper; in particular, projection is well-defined for (fully simplified) rational closed forms of PGFs.

reasoning and, in particular, in contrast to representations in terms of density or mass functions, the effective computation of (high-order) moments; (3) tail bounds, concentration bounds, and other properties of interest can be extracted with relative ease from a PGF; and (4) expressions containing parameters are naturally supported.

3.2 Semantics with Conditioning

We lift the approach to discrete, loopy probabilistic programs *with conditioning* by extending the PGF semantics of Klinkenberg et al. [2020] to cope with posterior observations. To define such a semantic model, we fix k $\mathbb{N}$-valued program variables $x_1, x_2, \ldots, x_k$. The set of program state valuations is $\mathbb{N}^k$; for each $\sigma = (\sigma_1, \ldots, \sigma_k) \in \mathbb{N}^k$, σ_i indicates the value of x_i. We consider the pGCL programming language [McIver and Morgan 2005] with the extended ability to specify posterior observations via the `observe` statements [Gordon et al. 2014; Nori et al. 2014; Olmedo et al. 2018]:

Definition 3 (cpGCL). *A program P in the* conditional probabilistic guarded command language (cpGCL) *adheres to the grammar*

$$P ::= \texttt{skip} \mid x := E \mid P \,;\, P \mid \{P\}\,[\,p\,]\,\{P\} \mid \texttt{observe}\,(B) \mid \texttt{if}\,(B)\,\{P\}\,\texttt{else}\,\{P\} \mid \texttt{while}\,(B)\,\{P\}$$

where $E\colon \mathbb{N}^k \to \mathbb{N}$ is an arithmetic expression, $B \subseteq \mathbb{N}^k$ is a predicate, and $p \in [0,1]$.[2]

The meaning of most cpGCL program constructs is standard. The *probabilistic choice* $\{P\}\,[\,p\,]\,\{Q\}$ executes P with probability $p \in [0,1]$ and Q with probability $1-p$. The *conditioning statement* $\texttt{observe}(B)$ "blocks" all program runs that violate the guard B and normalizes the probabilities of the remaining runs. For example, in Prog. 1 on page 2, the telephone operator observes 5 calls in the last hour as indicated by $\texttt{observe}\,(\,c = 5\,)$. To reflect this, all program states where $c \neq 5$ are assigned probability zero. The program's distribution is adjusted by normalizing the probability of runs satisfying $c = 5$ by the total probability mass of all runs violating this condition.

To identify program runs violating the observations, we extend the domain of FPSs – and thus the domain of PGFs – with a dedicated indeterminate $X_{\lightning}$ aggregating observation-violation probability:

Definition 4 (eFPS and ePGF). *Let $\mathbf{X}$ and $X_{\lightning}$ be indeterminates. For any program state valuations $\sigma \in \mathbb{N}^k$, an* extended *formal power series (eFPS) is of the form*[3]

$$F \;=\; [\lightning]_F X_{\lightning} + \sum\nolimits_{\sigma \in \mathbb{N}^k} [\sigma]_F \mathbf{X}^\sigma \quad \text{with} \quad [\cdot]_F\colon \mathbb{N}^k \cup \{\lightning\} \to \mathbb{R}_{\geq 0}\,.$$

We refer to $[\lightning]_F X_{\lightning}$ as the observation-violation term *and call the set of all extended formal power series* eFPS. *Let $|F| \triangleq \sum_{\sigma \in \mathbb{N}^k} [\sigma]_F$ denote the* mass *of F. $F \in$ eFPS is an* extended PGF (ePGF) *iff $|F| \leq 1$; in this case, F encodes a (sub)probability distribution. Let* ePGF *be the set of all* ePGFs. *An* ePGF transformer *is a function* ePGF $\to$ ePGF.

We emphasize that $|F|$ does not take the `observe`-violation probability $[\lightning]_F$ into account. Another way to obtain $|F|$ is through the substitution of indeterminates $\mathbf{X}$ representing program variables by $\mathbf{1}$ and the indeterminate $X_{\lightning}$ for the observation-violation by 0. Addition and scalar multiplication in eFPS are to be understood coefficient-wise, that is, for any $F, G \in$ eFPS,

$$F + G \;\triangleq\; \left([\lightning]_F + [\lightning]_G\right) X_{\lightning} + \sum\nolimits_{\sigma \in \mathbb{N}^k} \left([\sigma]_F + [\sigma]_G\right) \mathbf{X}^\sigma\,,$$

$$a \cdot F \;\triangleq\; \left(a[\lightning]_F X_{\lightning}\right) + \sum\nolimits_{\sigma \in \mathbb{N}^k} \left(a[\sigma]_F\right) \mathbf{X}^\sigma \quad \text{for} \quad a \in \mathbb{R}_{\geq 0}\,.$$

[2] We do not give an explicit syntax for E and B as it is irrelevant at this point. When dealing with *automation*, we present a concrete syntax, cf. Table 3 on page 13.

[3] The coefficients $[\cdot]_F$ range over $\mathbb{R}^\infty_{\geq 0}$ to enforce a complete lattice structure over eFPS; see details in [Klinkenberg et al. 2024a, Appx. A and B].

Table 2. The *non-normalized* semantics for cpGCL programs.

P	$[\![P]\!](G)$
`skip`	G
$x_i \coloneqq E$	$[\lightning]_G X_\lightning + \sum_\sigma [\sigma]_G X_1^{\sigma_1} \cdots X_i^{E(\sigma)} \cdots X_k^{\sigma_k}$
`observe` (B)	$([\lightning]_G + \lvert\langle G\rangle_{\neg B}\rvert)\, X_\lightning + \langle G\rangle_B$
$\{P_1\}\ [p]\ \{P_2\}$	$p \cdot [\![P_1]\!](G) + (1-p) \cdot [\![P_2]\!](G)$
`if` $(B)\ \{P_1\}$ `else` $\{P_2\}$	$[\lightning]_G X_\lightning + [\![P_1]\!](\langle G\rangle_B) + [\![P_2]\!](\langle G\rangle_{\neg B})$
$P_1 \fatsemi P_2$	$[\![P_2]\!]\big([\![P_1]\!](G)\big)$
`while` $(B)\ \{P_1\}$	$[\mathrm{lfp}\ \Phi_{B,P_1}]\,(G)$, where
	$\Phi_{B,P_1}(f) \;=\; \lambda G.\ [\lightning]_G X_\lightning + \langle G\rangle_{\neg B} + f\big([\![P_1]\!](\langle G\rangle_B)\big)$

Remark. eFPS is not closed under multiplication: $X_\lightning \cdot X_\lightning = X_\lightning^2 \notin$ eFPS. This is intended, as such monomial combinations do not have a valid interpretation in terms of probability distributions. ◁

We endow ePGFs with the following ordering relations.

Definition 5 (Orders over ePGF). *For all $F, G \in$ ePGF, let*

$$F \preceq G \qquad \text{iff} \qquad \forall \sigma \in \mathbb{N}^k \cup \{\lightning\}.\ [\sigma]_F \;\leq\; [\sigma]_G\,.$$

This order can be lifted to ePGF transformers, that is, for all $\phi, \psi \in$ (ePGF $\to$ ePGF),

$$\phi \sqsubseteq \psi \qquad \text{iff} \qquad \forall F \in \text{eFPS}.\ \phi(F) \preceq \psi(F)\,.$$

In fact, (ePGF, $\preceq$) and (ePGF $\to$ ePGF, $\sqsubseteq$) are ω-complete partial orders (cf. [Klinkenberg et al. 2024a, Appx. B]). To evaluate Boolean guards, we use the so-called *filtering* function for eFPSs. The filtering of $F \in$ eFPS by predicate B is

$$\langle F\rangle_B \;\triangleq\; \sum\nolimits_{\sigma \models B} [\sigma]_F \mathbf{X}^\sigma\,,$$

i.e., $\langle F\rangle_B$ is the eFPS derived from F by setting $[\lightning]_F$ and all $[\sigma]_F$ with $\sigma \not\models B$ to 0. In contrast to [Klinkenberg et al. 2020], we cannot decompose F into $\langle F\rangle_B + \langle F\rangle_{\neg B}$, but rather have to include the observation-violation term separately, yielding $F = \langle F\rangle_B + \langle F\rangle_{\neg B} + [\lightning]_F X_\lightning$. Further properties of the ePGF domain are found in [Klinkenberg et al. 2024a, Appx. B].

3.3 Non-Normalized Semantics for cpGCL

Let $[\![P]\!]$: ePGF $\to$ ePGF be a (non-normalized) distribution transformer for cpGCL program P. We define the *non-normalized* semantics of P by transforming an input ePGF G to an output ePGF $[\![P]\!](G)$ while *explicitly* keeping track of the probability of violating the observations; see Table 2.

The `skip` statement leaves the initial distribution G unchanged, i.e., it *skips* an instruction. The assignment $x_i \coloneqq E$ updates the exponent of the corresponding indeterminate X_i in every term of the ePGF by $E(\sigma)$ and the observation-violation term remains unchanged. For instance, given $E = 2 \cdot xy^3 + 23$ and state valuation $\sigma = (x, y) = (1, 10)$, $x_i \coloneqq E$ updatess the term aXY^{10} to $aX^{2023}Y^{10}$. The semantics for `observe`(B) is defined in line with [Bichsel et al. 2018; Jacobs 2021; Nori et al. 2014; Olmedo et al. 2018] as *rejection sampling*, i.e., if the current program run satisfies B, it behaves like a `skip` statement and the posterior distribution is unchanged; If the current run, however, violates the condition B, the run is rejected and the program restarts from the top in a reinitialized state. Hence, observing a certain guard B just filters the prior

distribution and accumulates the probability mass that violates the guard. For example, observing an even dice roll $\texttt{observe}\ (\ x \equiv_2 0\ )$ out of a six-sided die $\frac{1}{6}\left(X + X^2 + X^3 + X^4 + X^5 + X^6\right)$ yields $\frac{1}{6}\left(X^2 + X^4 + X^6\right) + \frac{1}{2}X_{\lightning}$. The probabilistic branching statement $\{\ P_1\ \}\ [\ p\ ]\ \{\ P_2\ \}$ is interpreted as the convex p-weighted combination of the two subprograms P_1 and P_2. The semantics of conditional branching $\texttt{if}\ (\ B\ )\ \{\ P_1\ \}\ \texttt{else}\ \{\ P_2\ \}$ combines the semantics of P_1 and P_2 conditionally based on B. Sequential composition $P_1 \fatsemi P_2$ composes programs in a *forward* manner, i.e., we first evaluate P_1 and take the intermediate result as new input for P_2. The semantics of a loop $\texttt{while}\ (\ B\ )\ \{\ P_1\ \}$ is defined as the *least fixed point* (lfp) of Φ_{B,P_1} (see domain theory in [Klinkenberg et al. 2024a, Appx. A]). Here, Φ_{B,P_1} is known as the *characteristic function* – a monotonic operator mimicking the effect of unfolding the loop. Concretely, Φ_{B,P_1} guarantees the equivalence of $\texttt{while}\ (\ B\ )\ \{\ P_1\ \}$ and $\texttt{if}\ (\ B\ )\ \{\ P_1 \fatsemi \texttt{while}\ (\ B\ )\ \{\ P_1\ \}\ \}\ \texttt{else}\ \{\ \texttt{skip}\ \}$.

Note that the $\texttt{observe}$-violation term $[\lightning]_G X_{\lightning}$ "passes through" all instructions but $\texttt{observe}\ (\ B\ )$:

Lemma 6 (Error Term Pass-Through). *For every program P and every $F \in$ ePGF,*

$$[\![P]\!](F) \;=\; [\![P]\!]\left(\sum\nolimits_{\sigma\in\mathbb{N}^k} [\sigma]_F \mathbf{X}^\sigma + [\lightning]_F X_{\lightning}\right) \;=\; [\![P]\!]\left(\sum\nolimits_{\sigma\in\mathbb{N}^k} [\sigma]_F \mathbf{X}^\sigma\right) + [\lightning]_F X_{\lightning}\ .$$

This renders the semantics as a conservative extension to [Chen et al. 2022a], as for $\texttt{observe}$-free programs on initial distributions without $[\lightning]_G X_{\lightning}$, both semantics coincide.

Recall that in Prog. 3 on page 3, all program runs which eventually would terminate violate the observation. Since the (unnormalized) probability of non-termination is zero (as there is only a single infinite run), the final *non-normalized, conditioned* ePGF semantics of this program is $1 \cdot X_{\lightning}$.

3.4 Normalized Semantics for cpGCL

The non-normalized semantics serves as an intermediate result to achieve our *normalized* semantics, which further addresses *normalization* of distributions.

Definition 7 (Normalization). *The* normalization operator *norm is a partial function defined as*[4]

$$\mathit{norm}\colon \text{ePGF} \rightharpoonup \text{ePGF}, \qquad F \mapsto \begin{cases} \frac{\langle F\rangle_{\texttt{true}}}{1-[\lightning]_F} & \text{if } [\lightning]_F < 1\,, \\ \text{undefined} & \text{otherwise}\,. \end{cases}$$

Intuitively, normalizing an ePGF amounts to "distributing" the probability mass $[\lightning]_F$ pertaining to observation violations over its remaining (valid) program runs. We lift the operator and denote the *normalized* semantics of P by

$$\mathit{norm}\left([\![P]\!]\right) \;\triangleq\; \lambda G.\ \mathit{norm}\left([\![P]\!](G)\right) \;=\; \lambda G.\ \frac{\langle [\![P]\!](G)\rangle_{\texttt{true}}}{1 - [\lightning]_{[\![P]\!](G)}}\,, \qquad \text{provided } [\lightning]_{[\![P]\!]_G} < 1.$$

Remark. In contrast to the non-normalized semantics, the normalized semantics might not always be defined: Reconsider Prog. 3 for which the non-normalized semantics is $1 \cdot X_{\lightning}$; normalizing the semantics is not possible as it would lead to $\frac{\langle 1\cdot X_{\lightning}\rangle_{\texttt{true}}}{1-[\lightning]_F} = \frac{0}{0}$, i.e., an undefined expression. This phenomenon can only be caused by $\texttt{observe}$-violations but never by non-terminating behaviors. The following two programs reveal the difference between non-termination and $\texttt{observe}$ violation: $\{\ x \coloneqq 1\ \}\ [\ 1/2\ ]\ \{\ \texttt{observe}\ (\ \texttt{false}\ )\ \}$ has a normalized semantics of $1 \cdot X^1$, whereas the normalized semantics for $\{\ x \coloneqq 1\ \}\ [\ 1/2\ ]\ \{\ \texttt{diverge}\ \}$[5] is $\frac{1}{2} \cdot X^1$. ◁

Example 8 (Telephone Operator). Reconsider Prog. 1, the loop-free program generating an infinite-support distribution. It describes a telephone operator who lacks knowledge about whether today is a weekday or weekend. The operator's initial belief is that there is a probability of $5/7$ of it

[4] *norm* in fact maps an ePGF to a PGF, i.e., $[\lightning]_F X_{\lightning}$ is pruned away by normalization.

[5] $\texttt{diverge}$ is syntactic sugar for $\texttt{while}\ (\ \texttt{true}\ )\ \{\ \texttt{skip}\ \}$.

being a weekday ($w = 0$) and a $2/7$ probability of it being a weekend ($w = 1$). Typically, on weekdays, there are an average of 6 incoming calls per hour, while on weekends, this rate decreases to 2 calls. Both rates are governed by a Poisson distribution. The operator has observed 5 calls in the past hour, and the objective is to determine the updated distribution of the initial belief based on this posterior observation. We start the computation with prior distribution (ePGF) 1, which initializes every program variable to 0 with probability 1. For the assignments to c we use the closed-form PGF for a Poisson distribution with parameter λ, which is given by $\sum_{k\in\mathbb{N}_0} \frac{\lambda^k e^{-\lambda}}{k!} C^k = e^{-\lambda} \sum_{k\in\mathbb{N}_0} \frac{(\lambda C)^k}{k!} = e^{-\lambda} e^{\lambda C} = e^{-\lambda(1-C)}$. By computing the transformations forward in sequence for each program instruction (see Prog. 6), we obtain the *non-normalized* semantics:

$$[\![P]\!](G) \;=\; \tfrac{(4860+8e^4W)}{105e^6} C^5 + (1 - \tfrac{4860+8e^4}{105e^6}) X_{\lightning} \;.$$

Normalizing this yields

$$\mathit{norm}\big([\![P]\!](G)\big) \;=\; \frac{(1215e^{-4} + 2W)C^5}{2 + 1215e^{-4}} \;. \qquad \triangleleft$$

```
/// 1                    (= 1 · W^0 C^0 + 0 · X_↯)
{ w := 0 } [5/7] { w := 1 } ;
/// 5/7 W^0 + 2/7 W^1
if ( w = 0 ) {
  /// 5/7
  c := poisson (6)
  /// 5/7 e^{-6(1-C)}
} else {
  /// 2/7 W
  c := poisson (2)
  /// 2/7 e^{-2(1-C)} W
} ;
/// 5/7 e^{-6(1-C)} + 2/7 e^{-2(1-C)} W
observe ( c = 5 )
/// (4860+8e^4 W)/105e^6 C^5 + (1 − (4860+8e^4)/105e^6) X_↯
```

Prog. 6. Semantics for the tel. operator.

Notably, the semantics in Table 2 coincides with an operationally modeled semantics using *countably infinite Markov chains* [Olmedo et al. 2018] – which in turn, for universally almost-surely terminating programs[6] is equivalent to the interpretation of Microsoft's probabilistic programming language R2 [Nori et al. 2014]. A Markov chain describing the semantics of a cpGCL program consists of three ingredients: (1) the state space $\mathcal{S}$, (2) the initial state $\langle P, \sigma\rangle$, and (3) a transition matrix $\mathcal{P} : \mathcal{S} \times \mathcal{S}$. The states are pairs of the form $\langle P, \sigma\rangle$. Here, P denotes the program left to be executed (with $\downarrow$ indicating the terminated program) and σ the current state valuation. We use the dedicated state $\langle \lightning \rangle$ for denoting that some observe violations have occurred during the run of a program. The detailed construction of the Markov chain $\mathcal{R}_\sigma[\![P]\!]$ from a cpGCL program P with initial state valuation σ is given in [Klinkenberg et al. 2024a, Appx. B]. Regarding the equivalence between the two semantics, we are interested in the reachability probability of eventually reaching state $\langle \downarrow, \sigma\rangle$ conditioned to never visiting the observe-violation state $\langle \lightning \rangle$.

Theorem 9 (Equivalence of Semantics). *For every* cpGCL *program P, let $\mathcal{R}_\sigma[\![P]\!]$ be the Markov chain of P starting with state valuation $\sigma \in \mathbb{N}^k$. Then, for any $\sigma' \in \mathbb{N}^k$,*

$$\Pr^{\mathcal{R}_\sigma[\![P]\!]}\big(\Diamond\langle\downarrow, \sigma'\rangle \mid \neg\Diamond\langle\lightning\rangle\big) \;=\; [\sigma']_{\mathit{norm}([\![P]\!](\mathbf{X}^\sigma))} \;, \tag{3}$$

where the left term denotes the probability of eventually reaching the terminating state $\langle\downarrow, \sigma'\rangle$ in $\mathcal{R}_\sigma[\![P]\!]$ conditioned on avoiding the observe*-failure state $\langle\lightning\rangle$.*

The coincidence captured in Eq. (3) ensures the adequateness of our ePGF semantics for cpGCL programs, which includes the case of *undefined semantics*, i.e., the conditional probability (LHS) is not defined if and only if the normalized semantics (RHS) is undefined. Again, for pGCL programs *without conditioning*, the *conditioned* semantic model is equivalent to that of [Klinkenberg et al. 2020] and thereby [Kozen 1981; McIver and Morgan 2005], since an observe-free program never induces the violation term $[\lightning]_F X_{\lightning}$ and hence, the *norm* operator has no effect.

[6] Programs that terminate with probability 1 on all inputs; see Section 4.2.

Table 3. Syntax (left) and the *non-normalized* semantics (right) of cReDiP programs.

P	$[\![P]\!](G)$
$x \coloneqq n$	$G[X_{\lightning}/0, X/1] \cdot X^n + (G - G[X_{\lightning}/0])$
$x{-}{-}$	$(G - G[X/0])X^{-1} + G[X/0]$
$x \mathrel{+}= \texttt{iid}\,(D, y)$	$G[X_{\lightning}/0, Y/Y[\![D]\!][T/X]] + (G - G[X_{\lightning}/0])$
$\texttt{if}\ (\ x < n\ )\ \{\ P_1\ \}\ \texttt{else}\ \{\ P_2\ \}$	$[\![P_1]\!](G_{x<n}) + [\![P_2]\!](G - G_{x<n})$, where $G_{x<n} = \sum_{i=0}^{n-1} \frac{1}{i!}(\partial_X^i G[X_{\lightning}/0])[X/0] \cdot X^i$
$P_1 \fatsemi P_2$	$[\![P_2]\!]([\![P_1]\!](G))$
$\texttt{while}\ (\ x < n\ )\ \{\ P_1\ \}$	$(\mathrm{lfp}\,\Phi_{x<n,P_1})(G)$, where $\Phi_{x<n,P_1}(\psi) = \lambda F.\ (F - F_{x<n}) + \psi([\![P_1]\!](F_{x<n}))$
$\texttt{observe}\ (\ \texttt{false}\ )$	$G[\mathbf{X}/\mathbf{1}, X_{\lightning}/1] \cdot X_{\lightning}$

4 EXACT BAYESIAN INFERENCE WITH LOOPS

Loops significantly complicate inferring posterior distributions of probabilistic programs. Computing the exact least fixed point of the characteristic function $\Phi_{B,P}$ (see Table 2) is in general highly intractable, and other techniques like *invariant*-based reasoning are used. Given the loop $\texttt{while}\ (\ B\ )\ \{\ P\ \}$, we call an ePGF transformer $I\colon \text{ePGF} \to \text{ePGF}$ an *invariant* if $\Phi_{B,P}(I) = I$, i.e., it remains unchanged when pushed through one loop iteration.

Effectively, reasoning about loops is reduced to *two challenges*: (1) finding an invariant candidate I, and (2) verifying that I is indeed a valid invariant, i.e., deciding whether $\Phi_{B,P}(I) = I$. Since the semantics of a program is also of type ePGF $\to$ ePGF, we can describe such an invariant by means of a program. To facilitate reasoning about such loop invariant programs, we consider a restricted set of cpGCL programs, called cReDiP. We further extend the program semantics to second-order ePGFs (eSOPs) to enable reasoning about multiple input distributions simultaneously. We develop an eSOP-based equivalence checking technique for cReDiP programs to reason about loop invariants in a *non-normalized* semantics. This technique also enables invariant synthesis by solving equation systems yielding parameter values satisfying the invariant condition $\Phi_{B,P}(I) = I$.

4.1 Program Equivalence

Checking whether a loop-free program I is an invariant of $\texttt{while}\ (\ B\ )\ \{\ P\ \}$ amounts to checking whether $\Phi_{B,P}([\![I]\!]) = [\![I]\!]$. Phrased in terms of generating functions, this reads

$$\forall G \in \text{ePGF}.\ \forall \sigma \in \mathbb{N}^k \cup \{\lightning\}.\quad [\sigma]_{\Phi_{B,P}([\![I]\!])(G)} \;=\; [\sigma]_{[\![I]\!](G)}\ . \tag{4}$$

Namely, we need to check the equivalence of two loop-free programs. As program equivalence is undecidable in general, we introduce a syntactic fragment of cpGCL called cReDiP (conditional rectangular discrete probabilistic programs) for which equivalence of loop-free programs is decidable.

The cReDiP *language.* Table 3 describes the syntax and semantics of cReDiP. This fragment contains multiple statements to update the values of program variables. Intuitively, the updates are performed by extracting the parts of the ePGF that are affected by the update through substitution operations. For example, $x \coloneqq n$ drops the observation-violation term and marginalizes w.r.t. X (thus effectively setting x to 0 temporarily) and then performs a shift by n in X. Finally, the unaffected part of the ePGF is added back to complete the transformation.

A prominent difference to pGCL is the statement $x \mathrel{+}= \texttt{iid}\,(D, y)$. Intuitively, it can be interpreted as a bounded loop, namely $\texttt{loop}(y)\{x \mathrel{+}= \texttt{sample}(D)\}$ where the number of iterations is given

by program variable y. More specifically, $x \mathrel{+}= \texttt{iid}\,(D, y)$ combines a series of operations: First independently sample y many random variables from distribution D and second, sum up the sampled values and increment x by that amount. For example, the program $P \coloneqq y \coloneqq 10; x \coloneqq 0; x \mathrel{+}= \texttt{iid}\,(\texttt{bernoulli}\,(1/2), y)$ describes a binomial distribution in X with parameters $n = 10$ and $p = 1/2$, i.e. $[\![P]\!] = Y^{10} \cdot (1/2 + 1/2X)^{10}$.

Moreover, we emphasize that Boolean guards in cReDiP can only be of the form $x < n$ where $n \in \mathbb{N}$ is a constant. We denote by $G_{x<n}$ the PGF G restricted to its terms with low enough order satisfying the guard $x < n$. The required elements of the PGF are collected by constructing the i-th formal derivative (for every $0 \leq i < n$) w.r.t. X and extracting the constant monomials (in X), i.e. the coefficients of monomial X^i in G. By nesting of `if`-statements, axis-aligned hyper-rectangles can be identified, i.e., in this way we can express conjunction, disjunction and negation of guards. The latter enables us to only consider `observe(false)` statements in our syntax, as we can reconstruct the full "rectangular" expressiveness for `observe` statements.

The key feature of the cReDiP language is that its loop-free fragment preserves rational closed-form ePGF representations; see Table 3 and [Chen et al. 2022a]. Hence, we can effectively compute the semantics of a loop-free cReDiP program given *one* closed-form representation of the input distribution. However, in order to decide program equivalence per Eq. (4), we need to compute the semantics of *infinitely or even uncountably many* input distributions. Chen et al. [2022a] solved this issue by introducing second-order PGFs; intuitively, these are FPS whose coefficients themselves are PGFs. We extend this idea for programs with conditioning:

Definition 10 (Second-Order ePGF). *Let* $\mathbf{U} = (U_1, \ldots, U_k)$ *be a tuple of formal indeterminates, that are pairwise distinct from* $\mathbf{X} = (X_1, \ldots, X_k)$ *and* $X_\lightning$ *of* eFPS. *A* second-order *ePGF is a generating function of the form*

$$G = \sum_{\sigma \in \mathbb{N}^k} G_\sigma U^\sigma = \sum_{\sigma \in \mathbb{N}^k} (\langle G_\sigma \rangle_{\texttt{true}} + [\lightning]_{G_\sigma} X_\lightning) U^\sigma = \sum_{\sigma \in \mathbb{N}^k} \langle G_\sigma \rangle_{\texttt{true}} U^\sigma \;+\; \sum_{\sigma \in \mathbb{N}^k} [\lightning]_{G_\sigma} X_\lightning U^\sigma,$$

where $G_\sigma \in$ ePGF. *We denote the set of second-order ePGFs by* eSOP.

An eSOP hence represents, in a single formal power series, *multiple* ePGFs as coefficients G_σ of different monomials $\mathbf{U}^\sigma$. Intuitively one can interpret $\mathbf{U}$ as eFPS formal indeterminates of additional program variables which do not occur in the program and whose sole purpose is to remember the actual program variables' initial values. We can naturally extend the denotational semantics described in Table 3 to eSOP, as demonstrated by the following example.

Example 11 (eSOP Semantics of cReDiP Program). Consider the cReDiP program P in Prog. 7 together with the eSOP input generating function $G = 1X^1 \cdot U^1 + 1X^2 \cdot U^2 + 1X^3 \cdot U^3$, identifying indeterminate X and meta-indeterminate U for program variable x. This eSOP represents three Dirac distributions, i.e., $1X^1$, $1X^2$, and $1X^3$, where the purpose of U is to remember the initial value of x. We now examine the computation of $[\![P]\!](G)$ step-by-step, starting with the increment operation which only affects the indeterminate X of the involved program variable x and does not affect U. To this end, we substitute $(\frac{1}{2} + \frac{1}{2}X) \cdot X$ for X, since G contains no initial observation-violation term. Afterwards, to aggregate the states that violate the observation, the semantics also substitutes 1 for

$/\!/\!/\; X^1 \cdot U^1 + X^2 \cdot U^2 + X^3 \cdot U^3$
$x \mathrel{+}= \texttt{iid}\,(\texttt{bernoulli}\,(1/2), x)$
$/\!/\!/\; \frac{1}{2}(X^1 + X^2) \cdot U^1 + \frac{1}{4}(X^2 + 2X^3 + X^4) \cdot U^2 + \frac{1}{8}(X^3 + 3X^4 + 3X^5 + X^6) \cdot U^3$
$\texttt{observe}\,(x < 3)$
$/\!/\!/\; \frac{1}{2}(X^1 + X^2) \cdot U^1 + \frac{1}{4}(X^2 + 3X_\lightning) \cdot U^2 + X_\lightning \cdot U^3$

Prog. 7. A cReDiP program annotated with eSOP semantics.

indeterminate X (and $X_{\lightning}$) and leaves the meta-indeterminates unchanged. As a result, we obtain $[\![P]\!](G) = 1/2(X^1+X^2)\cdot U^1 + 1/4(X^2+3X_{\lightning})\cdot U^2 + X_{\lightning}\cdot U^3$, and have computed all posterior distributions for initial state valuations $x=1, x=2, x=3$ in one shot. For instance, when starting with initial distribution $1X$ the posterior distribution is $1/2(X^1+X^2)$ as indicated by the coefficient of U^1. Finally, we note that the meta-indeterminates just "pass through" the eSOP semantics functional, i.e., it can be seen as the point-wise lifting of the ePGF semantics. ◁

Theorem 12 (eSOP Semantics). *Let P be a loop-free* cReDiP *program. Let $G = \sum_{\sigma\in\mathbb{N}^k} G_\sigma \mathbf{U}^\sigma \in$ eSOP. The* eSOP *semantics $[\![P]\!]$: eSOP $\to$ eSOP of P can be computed by*

$$[\![P]\!](G) \;=\; \sum\nolimits_{\sigma\in\mathbb{N}^k} [\![P]\!](G_\sigma)\cdot \mathbf{U}^\sigma \,.$$

Since PGF semantics is an instance of the general framework of Kozen's measure transformer semantics [Klinkenberg et al. 2020; Kozen 1981], the posterior distribution of a cReDiP program P is uniquely determined by its semantics on all possible Dirac distributions. One can thus construct an eSOP from P that represents all possible point-mass distributions for the program variables:

Definition 13 (Equivalence-Witness eSOP). *Let $\hat{G}$ be an eSOP defined as*

$$\hat{G} \;\triangleq\; \underbrace{(1-X_1U_1)^{-1}\cdots(1-X_kU_k)^{-1}}_{\textit{rational closed form}} \;=\; \sum\nolimits_{\sigma\in\mathbb{N}^k} \mathbf{X}^\sigma\mathbf{U}^\sigma \;=\; 1 + (1\mathbf{X})\mathbf{U} + (1\mathbf{X}^2)\mathbf{U}^2 + \cdots \,,$$

where the meta-indeterminates $\mathbf{U}$ *serve the purpose of "remembering" the initial state valuations.*

For the purpose of deciding program equivalence, $\hat{G}$ is particularly useful, since it represents Dirac distributions for all potential initial state valuations, with the exception of any observe-violation probabilities. This is, however, not a problem, as such observation-violation terms can be immediately removed from the equivalence check (by Lemma 6). As a consequence, we can use $\hat{G}$ to characterize program equivalence of loop-free cReDiP programs using eSOP. This is expressed by the following lemma.

Lemma 14 (eSOP Characterization). *Let P_1 and P_2 be loop-free* cReDiP *programs with $Vars(P_i) \subseteq \{x_1,\ldots,x_k\}$ for $i\in\{1,2\}$. Further, consider a vector $\mathbf{U} = (U_1,\ldots,U_k)$ of meta-indeterminates. Then,*

$$\forall G \in \text{ePGF}.\;\; [\![P_1]\!](G) \;=\; [\![P_2]\!](G) \qquad \text{iff} \qquad [\![P_1]\!](\hat{G}) \;=\; [\![P_2]\!](\hat{G}).$$

As we can compute $[\![P]\!](\hat{G})$ for loop-free $P \in$ cReDiP, the following consequence is immediate.

Corollary 15 (Decidability of Equivalence). *Let P_1, P_2 be two loop-free* cReDiP *programs. Then,*

$$\forall G \in \text{ePGF}.\;\; [\![P_1]\!](G) \;=\; [\![P_2]\!](G) \quad \textit{is decidable.}$$

Proof. By utilizing Lemma 14, we can rephrase the problem of determining program equivalence through the eSOP characterization $[\![P_1]\!](\hat{G}) = [\![P_2]\!](\hat{G})$. It is worth noting that $\hat{G}$ represents a *rational closed-form* eSOP $\hat{G} = \frac{1}{1-X_1U_1}\frac{1}{1-X_2U_2}\cdots\frac{1}{1-X_kU_k} \in \mathbb{R}[[\mathbf{X},\mathbf{U}]]$. For our purposes, we can disregard the portion of $\hat{G}$ that describes the initial observe violation behavior, as it immediately cancels out (see Lemma 6). As $\hat{G}$ is in rational closed form, both $[\![P_1]\!](\hat{G})$ and $[\![P_2]\!](\hat{G})$ must also possess a rational closed form since loop-free cReDiP semantics preserve closed forms; see Table 3 and [Chen et al. 2022a]. Additionally, the effective computation of $[\![P_1]\!](\hat{G}) = F_1/H_1$ and $[\![P_2]\!](\hat{G}) = F_2/H_2$ is possible because both P_1 and P_2 are loop-free programs.

In $\mathbb{R}[[\mathbf{X}, X_{\lightning}, \mathbf{U}]]$, the question of whether two formal power series represented as rational closed forms, namely F_1/H_1 and F_2/H_2, are equal can be decided:

$$\frac{F_1}{H_1} \;=\; \frac{F_2}{H_2} \qquad\Longleftrightarrow\qquad F_1H_2 \;=\; F_2H_1,$$

since the latter equation concerns the equivalence of two polynomials in $\mathbb{R}[\mathbf{X}, X_{\lightning}, \mathbf{U}]$. Therefore, we can compute these two polynomials and verify whether their (finite number of) non-zero coefficients coincide. If they do, then P_1 and P_2 are equivalent (i.e., $[\![P_1]\!] = [\![P_2]\!]$), whereas if they do not, they are not equivalent. In the case of non-equivalence, we can generate a Dirac distribution that produces two distinct outcomes. This is achieved by taking the difference $F_1H_2 - F_2H_1$ and computing the first non-zero coefficient in $\mathbb{R}[\mathbf{X}, X_{\lightning}]$. Then, extracting the exponent of the monomial describes an initial state valuation σ, with $[\![P_1]\!](\sigma) \neq [\![P_2]\!](\sigma)$. □

Remark. The proof of Corollary 15 (on decidability of equivalence) relies on the fact that the eSOP transformer $[\![P]\!](\cdot)$ preserves rational closed-form eSOPs. cReDiP is a non-trivial fragment of cpGCL for which we can show the preservation of rational closed forms for loop-free programs; but it is not necessarily the largest class of programs that features such a property. Investigating a more expressive fragment with decidability of equivalence is subject to future work. ◁

4.2 Invariant-Based Reasoning with Conditioning

cReDiP is a fragment of cpGCL for which the equivalence of loop-free programs is decidable. We now exploit this result to reason about loops in cReDiP programs. The key idea is to use loop-free cReDiP programs as potential invariant candidates. Recall the two main challenges of invariant-based reasoning: first, find an invariant candidate, and second, verify that it is indeed an invariant, i.e., $\Phi_{B,P}(I) = I$. In the remainder of this section, we focus on *verifying* invariant candidates given in the form of cReDiP programs, while deferring finding invariants to Section 5.

We first introduce the notion of lossless ePGF transformers to capture program termination:

Definition 16 (Lossless ePGF Transformers). *An ePGF transformer* $H\colon \mathsf{ePGF} \to \mathsf{ePGF}$ *is* lossless *for* $F \in \mathsf{ePGF}$ *if*

$$|H(F)| + [\lightning]_{H(F)} \;=\; |F| + [\lightning]_F \,.$$

H is universally lossless *if it is lossless for all F in* ePGF.

Intuitively, a lossless ePGF transformer is a mapping that does not leak any probability mass. Since the semantics of a program P is an ePGF transformer, $[\![P]\!]$ being (universally) lossless coincides with P being (universally) almost-surely terminating, abbreviated as (U)AST [Bournez and Garnier 2005; Saheb-Djahromi 1978]. Given $L = \texttt{while}\,(\,B\,)\,\{\,P\,\}$, we can approximate its least fixed point lfp $\Phi_{B,P}$ leveraging domain theory, in particular, Park's lemma, namely, $\Phi_{B,P}(I) \sqsubseteq I$ implies $[\![L]\!] \sqsubseteq I$ [Park 1969]. It enables reasoning about while-loops in terms of over-approximations and – in case a program is UAST– also about program equivalence.

Theorem 17 (Loop Invariants). *Given* $L = \texttt{while}\,(\,B\,)\,\{\,P\,\}$ *and a universally lossless ePGF transformer* $I\colon \mathsf{ePGF} \to \mathsf{ePGF}$. *We have*

(1) If $\Phi_{B,P}(I) \sqsubseteq I$, *then* $\mathit{norm}([\![L]\!](F)) \preceq \mathit{norm}(I(F))$ *whenever* $\mathit{norm}(I(F))$ *is defined.*
(2) If L is UAST, then I is an invariant of L if and only if

$$[\![L]\!] \;=\; I \qquad \text{and} \qquad \mathit{norm}([\![L]\!](F)) \;=\; \mathit{norm}(I(F)) \,.$$

Proof. For (1), we first prove that the normalization function is monotonic, whenever it is defined. Let $F, G \in \mathsf{ePGF}$ such that $\mathit{norm}(F)$, $\mathit{norm}(G)$ are defined. We have:

$$\begin{aligned} F \preceq G \quad &\Longrightarrow \quad [\lightning]_F \leq [\lightning]_G \quad \text{and} \quad \sum_{\sigma\in\mathbb{N}^k} [\sigma]_F \mathbf{X}^\sigma \preceq \sum_{\sigma\in\mathbb{N}^k} [\sigma]_G \mathbf{X}^\sigma \\ &\Longrightarrow \quad 1 - [\lightning]_F \geq 1 - [\lightning]_G \quad \text{and} \quad \sum_{\sigma\in\mathbb{N}^k} [\sigma]_F \mathbf{X}^\sigma \preceq \sum_{\sigma\in\mathbb{N}^k} [\sigma]_G \mathbf{X}^\sigma \end{aligned}$$

```
while ( y = 1 ) {
  { y ≔ 0 } [ 1/2 ] { y ≔ 1 } ;
  x ≔ x + 1 ;
  observe ( x < 3 ) }
```

Prog. 8. A truncated geometric distribution generator.

```
if ( y = 1 ) {
  x += iid ( geom (1/2) , y ) ;
  y ≔ 0 ;
  observe ( x < 3 ) }
```

Prog. 9. A loop-free cReDiP invariant of Prog. 8.

$$\begin{aligned}
&\implies && \frac{1}{1-[\lightning]_F} \le \frac{1}{1-[\lightning]_G} \quad \text{and} \quad \sum_{\sigma\in\mathbb{N}^k} [\sigma]_F \mathbf{X}^\sigma \le \sum_{\sigma\in\mathbb{N}^k} [\sigma]_G \mathbf{X}^\sigma \\
&\implies && \frac{1}{1-[\lightning]_F} \cdot \sum_{\sigma\in\mathbb{N}^k} [\sigma]_F \mathbf{X}^\sigma \le \frac{1}{1-[\lightning]_G} \cdot \sum_{\sigma\in\mathbb{N}^k} [\sigma]_G \mathbf{X}^\sigma \\
&\implies && \mathit{norm}(F) \le \mathit{norm}(G) \,.
\end{aligned}$$

It follows that $\mathit{norm}(\llbracket \texttt{while}\,(B)\,\{P\} \rrbracket(F)) \le \mathit{norm}(I(F))$, due to Park's lemma.

For (2), first assume that $\llbracket L \rrbracket = I$ and $\mathit{norm}(\llbracket L \rrbracket(F)) = \mathit{norm}(I(F))$. As $I = \llbracket L \rrbracket = \mathrm{lfp}\,\Phi_{B,P}$, I is trivially identified as an invariant. For the other direction, assume that I is an invariant (i.e., a fixed point). Thus, I must be at least $\mathrm{lfp}\,\Phi_{B,P} = \llbracket \texttt{while}\,(B)\,\{P\} \rrbracket$. Moreover, because $\texttt{while}\,(B)\,\{P\}$ is UAST, it follows that

$$\left|\llbracket \texttt{while}\,(B)\,\{P\} \rrbracket(F)\right| + [\lightning]_{\llbracket \texttt{while}(B)\{P\} \rrbracket(F)} = |F| + [\lightning]_F = |I(F)| + [\lightning]_{I(F)} \quad \text{for all } F \in \mathsf{ePGF}\,.$$

The second equality arises from I being universally lossless. Combining these results yields

$$\begin{aligned}
\forall F \in \mathsf{ePGF}.\ \big(&\llbracket \texttt{while}\,(B)\,\{P\} \rrbracket(F) \le I(F) \\
&\text{and } \left|\llbracket \texttt{while}\,(B)\,\{P\} \rrbracket(F)\right| + [\lightning]_{\llbracket \texttt{while}(B)\{P\} \rrbracket(F)} = |I(F)| + [\lightning]_{I(F)}\big) \\
&\implies \forall F \in \mathsf{ePGF}.\ \llbracket \texttt{while}\,(B)\,\{P\} \rrbracket(F) = I(F) \iff \llbracket \texttt{while}\,(B)\,\{P\} \rrbracket = I\,.
\end{aligned}$$

Then, $\mathit{norm}(\llbracket \texttt{while}\,(B)\,\{P\} \rrbracket(F)) = \mathit{norm}(I(F))$ follows for all $F \in \mathsf{ePGF}$. □

Combining the results from this section, we can state the decidability of checking invariant validity for loop-free cReDiP candidates.

Theorem 18. *Let $L = \texttt{while}\,(B)\,\{P\} \in$ cReDiP be UAST with loop-free body P and I be a loop-free* cReDiP *program. It is decidable whether $\llbracket L \rrbracket = \llbracket I \rrbracket$.*

Proof. The correctness is an immediate consequence of Theorem 17 and Corollary 15. □

We demonstrate our invariant-based reasoning technique by Example 19.

Example 19 (Geometric Distribution Generator). Prog. 8 describes an iterative algorithm that repeatedly flips a fair coin – while counting the number of trials – until seeing heads, and observes that the number of trials is less than 3. Assume we want to compute the posterior distribution for input $1 \cdot Y^1 X^0$ (i.e. $y = 1$ and $x = 0$). We first evaluate $\mathrm{lfp}\,\Phi_{B,P}$. Using Theorem 17 (2), we perform an *equivalence check* on the invariant in Prog. 9. As Prog. 8 and 9 are equivalent, we substitute the loop-free program for the `while`-loop and continue. The resulting posterior distribution for input Y is $\llbracket P \rrbracket(Y) = \frac{4}{7} + \frac{2}{7}X + \frac{1}{7}X^2$. Since Prog. 8 is UAST, this is its *precise posterior distribution*. The step-by-step computation of the equivalence check can be found in [Klinkenberg et al. 2024a, Appx. E]. ◁

To summarize, *reasoning about program equivalence using eSOPs enables exact Bayesian inference for* cReDiP *programs containing loops*. We remark that *nested loops* can be treated in a compositional

manner: We first provide a loop-free invariant for the inner loop, prove its correctness (i.e., equivalence), and then replace the inner loop by its invariant and repeat the procedure for the outer loop. This feature of compositional reasoning is a key benefit of reusing the loop-free fragment of cReDiP as a specification language to describe invariants.

4.3 Equivalence of Normalized Semantics

Our previous notion of equivalence $[\![L]\!] = [\![I]\!]$ describes the equivalence of the *non-normalized* semantics, i.e., the `while`-loop and the loop-free invariant generate exactly the same distributions and `observe`-violation probabilities, which immediately entails also the equivalence of the normalized semantics, i.e., $\mathit{norm}([\![L]\!]) = \mathit{norm}([\![I]\!])$, but not necessarily the reverse. In practice, however, it is interesting to have a *weaker* notion of equivalence which addresses only the normalized semantics, regardless of observation violations (as programmers may use different observation strategies to construct programs yielding the same output distribution). This weaker notion reads as

$$P \sim Q \quad \text{iff} \quad \forall G \in \mathsf{PGF}.\ \mathit{norm}([\![P]\!](G)) = \mathit{norm}([\![Q]\!](G)) \ . \tag{5}$$

We aim to capture such equivalence again using eSOPs. First, we lift the operator *norm* to eSOPs:

Definition 20 (Conditioning on eSOP). *Let $G \in \mathsf{eSOP}$. The function*

$$\mathit{cond}\colon \mathsf{eSOP} \to \mathsf{SOP}\,, \qquad G \mapsto \sum\nolimits_{\sigma \in \mathbb{N}^k} \mathit{norm}(G_\sigma)\mathbf{U}^\sigma \ .$$

is called the conditioning function.

For simplicity, we assume that $\forall \sigma \in \mathbb{N}^k.\ G_\sigma \neq X_{\lightning}$ as otherwise *norm* is not defined. Note that *cond* often *cannot* be evaluated in a closed-form eSOP as there may be *infinitely many* ePGF coefficients of the (non-normalized) eSOP that have different observation-violation probabilities. However, we present a sufficient condition under which *cond* can be evaluated on closed-form eSOPs:

Proposition 21. *Let $F_1, F_2 \in \mathsf{ePGF}$, with $p \coloneqq [\lightning]_{F_1} = [\lightning]_{F_2}$. Then,*

$$\mathit{cond}(F_1) + \mathit{cond}(F_2) = \frac{\langle F_1 \rangle_{\mathtt{true}} + \langle F_2 \rangle_{\mathtt{true}}}{1-p} = \mathit{cond}(F_1 + F_2) \ .$$

Intuitively, addition distributes over *cond*, i.e., *cond* behaves linearly. Generalizing this concept to finitely many equal `observe`-violation properties we get the following.

Corollary 22 (Partitioning). *Let S be a finite partitioning of $\mathbb{N}^k = S_1 \uplus \cdots \uplus S_m$ with $[\lightning]_{G_\sigma} = [\lightning]_{G_{\sigma'}}$, for all $\sigma, \sigma' \in S_i,\ 1 \leq i \leq m$. Then:*

$$G \;=\; \sum\nolimits_{i=1}^{m} \sum\nolimits_{\sigma \in S_i} ([\lightning]_{S_i} X_{\lightning} + \langle G_\sigma \rangle_{\mathtt{true}})\mathbf{U}^\sigma \ ,$$

where $[\lightning]_{S_i}$ denotes the observation-violation probability in S_i. For such G we have:

$$\mathit{cond}(G) \;=\; \sum\nolimits_{i=1}^{m} \frac{\sum_{\sigma \in S_i} \langle G_\sigma \rangle_{\mathtt{true}} \mathbf{U}^\sigma}{1 - [\lightning]_{S_i}} \ .$$

Unfortunately, requiring a finite partitioning is quite restrictive. Finite partitioning is impossible already for some loop-free programs, an example is provided in Prog. 10. Given an initial distribution for variable y, the program computes the sum of y-many independent and identically distributed Bernoulli variables with success probability $1/2$. This is equivalent to sampling from a binomial distribution with y trials and probability $1/2$. Finally, it marginalizes the distribution by assigning y to zero and conditions on the event that x is less than 1, resulting in $\sum_{i=0}^{\infty} \frac{\left(2^{-i} + (1-2^{-i})X_{\lightning}\right)V^i}{(1-U)}$. We can deduce that for any initial state valuation (x, y) we obtain a *different* `observe` violation probability $(1 - 2^{-y})$, hence we cannot finitely partition the state space into equal violation probability classes.

$$/\!/\!/ (1-XU)^{-1}(1-YV)^{-1}$$
$$x \coloneqq 0;$$
$$/\!/\!/ (1-U)^{-1}(1-YV)^{-1}$$
$$x \mathrel{+}= \mathtt{iid}\,(\mathtt{bernoulli}\,(1/2)\,, y);$$
$$/\!/\!/ 2(1-U)^{-1}(2-(1+X)YV)^{-1}$$
$$y \coloneqq 0;$$
$$/\!/\!/ 2(1-U)^{-1}(2-(1+X)V)^{-1}$$
$$\mathtt{observe}\ (x < 1);$$
$$/\!/\!/ \frac{2(1-X_{\not 2})}{(1-U)(2-V)} + \frac{X_{\not 2}}{(1-U)(1-V)}$$
$$/\!/\!/ \sum_{i=0}^{\infty} \frac{\left(2^{-i} + (1-2^{-i})X_{\not 2}\right) V^i}{(1-U)}$$

Prog. 10. Program with infinitely many observe violation probabilities.

```
while ( n > 0 ) {
    { n := n − 1 } [ q/3 ] { c := c + 1 }
}
```

Prog. 11. n-geometric generator with success probability $q/3$ for $0 \le q \le 3$.

```
/* sums n geometric(p) samples */
c += iid (geom (p) , n);
/* on termination n is zero */
n := 0
```

Prog. 12. n-geometric invariant with parameter p.

Another challenge when considering the equivalence of normalized distributions is: Evaluating *cond* on (closed-form) eSOPs yields that $\mathit{cond}([\![P]\!](\hat{G})) = \mathit{cond}([\![Q]\!](\hat{G}))$. This implies $\forall \sigma \in \mathbb{N}^k.\ \mathit{norm}([\![P]\!](\mathbf{X}^\sigma)) = \mathit{norm}([\![Q]\!](\mathbf{X}^\sigma))$, i.e., equivalence on point-mass distributions. However, we do not necessarily have the precise equivalence as per Eq. (5), because the *norm* operator used to define *cond* is a non-linear function[7] and thus the point-mass distributions cannot be combined in a sensible way. However, in many use cases we are only interested in the behavior of a specific initial state valuation where such a result on point-mass equivalence can still be useful.

5 FINDING INVARIANTS USING PARAMETER SYNTHESIS

In contrast to the previous section which aims at *validating a given invariant*, in this section, we address the problem of *finding* such invariants. For related problems, e.g., finding invariants in terms of weakest preexpectations, there exist sound and complete synthesis algorithms for *subclasses* of loops and properties that can be verified by piecewise linear templates [Batz et al. 2023]. We adopt the idea of template-based invariant synthesis and leverage the power of eSOPs to achieve decidability results for a subclass of invariant candidates. Our templates are described by parametric loop-free cReDiP programs, e.g., $I_p = \{\, x \coloneqq 1 \,\}\ [\, p \,]\ \{\, x \coloneqq 0 \,\}$ which models a Bernoulli distribution with symbolic parameter p. We believe that (1) using programs as templates is (in particular in the probabilistic case) intuitively easier than using first-order logic as typically used to express invariants, and (2) finding suitable templates can be encoded as a program synthesis problem whose hardness may be precisely quantified. Recall the invariant synthesis problem: Given a `while`-loop $L = \mathtt{while}\ (\, B \,)\ \{\, P \,\}$, find a loop-free `cReDiP` program I such that $\Phi_{B,P}([\![I]\!]) = [\![I]\!]$. Sometimes, the general shape of an invariant template $T_{\mathbf{p}}$ (with a vector $\mathbf{p}$ of parameters) is derivable from L, but finding a valid parameter valuation may be involved. We illustrate the idea by Example 23.

Example 23 (n-Geometric Parameter Synthesis). Prog. 11 (with loop body P) is a variant of Prog. 2, where instead of requiring one success (setting $h = 0$), we need n successes to terminate. Furthermore, the individual success probability is $\frac{q}{3}$, where q is a symbolic parameter. It seems

[7] For the non-normalized semantics, general equivalence $[\![P]\!] = [\![Q]\!]$ follows from the linearity of the transformer.

natural that this program encodes the n-fold geometric distribution[8] with individual success probability $\frac{q}{3}$. This suggests to formulate the invariant template Q_p given in Prog. 12, where c is a sum of n geometric distributions with an unknown parameter p. Using Theorem 17, we can derive the equivalence of Prog. 11 and Prog. 12 and obtain an equation in p and q:

$$\Phi_{B,P}([\![Q_p]\!])(\hat{G}) = -\frac{(-3+qCU+3C-3pC-qU+3pU-3pCU)}{3(-1+CV)(-1+C-pC+pU)}$$

$$[\![Q_p]\!](\hat{G}) = -\frac{(-1+C-pC)}{(-1+CV)(-1+C-pC+pU)}$$

$$\text{Then} \qquad \Phi_{B,P}([\![Q_p]\!])(\hat{G}) = [\![Q_p]\!](\hat{G}) \qquad \text{iff} \qquad p = \frac{q}{3}\,.$$

The formal variable C corresponds to program variable c, while U and V are meta-indeterminates corresponding to the variables n and c. This result tells us, that for $p = \frac{q}{3}$ our parametrized invariant program is an invariant of Prog. 11. ◁

This approach works in general as the following theorem describes:

Theorem 24 (Decidability of Parameter Synthesis). *Let W be a* cReDiP `while` *loop and $I_{\mathbf{p}}$ be a parametrized loop-free* cReDiP *program. The problem whether there exist parameter values ρ such that the instantiated template I_ρ is an invariant, i.e.,*

$$\exists\, \mathbf{p} \in \mathbb{R}^I.\quad [\![W]\!] \;=\; [\![I_{\mathbf{p}}]\!] \qquad \textit{is decidable.}$$

Proof. Similar to Corollary 15. For full details, see [Klinkenberg et al. 2024a, Appx. D]. □

Note that in this formulation, parameters may depend on other parameters, but are always *independent* of all program variables and second-order indeterminates. Unfortunately, not every parametric invariant can be expressed by a loop-free cReDiP program as illustrated by the following example.

Example 25 (Hypergeometric Invariant). Prog. 13 encodes a biased 2-dimensional bounded random walk. In each turn, it decrements one of the variables with equal probability 1/2 until either the value of m or n arrives at 0. For any fixed program state valuation $(0,0) \neq (m,n) \in \mathbb{N}^2$, the number of loop iterations is bounded by $n+m-1$. We are interested in the exact posterior distribution for arbitrary input distributions. Due to its finite nature for any particular input distribution with finite support, we can analyze this program automatically using Prodigy by unfolding the loop $m+n-1$ times. For instance, the resulting distribution for an initial Dirac distribution describing the state valuation (a,b), is $[\![P]\!](M^a N^b) = \sum_{i=1}^{a} \frac{M^i}{2^{a+b-i}} \cdot \binom{a+b-i-1}{b-1} + \sum_{i=1}^{b} \frac{N^i}{2^{a+b-i}} \cdot \binom{a+b-i-1}{a-1}$. Using the simplification function in Mathematica [Inc. 2023], we derive the closed form,

```
while ( n > 0 ∧ m > 0 ) {
    { m := m − 1 } [ 1/2 ] { n := n − 1 }
}
```

Prog. 13. Dependent negative binomial variables.

$$I(a,b) = 2^{1-a-b} M \binom{-2+a+b}{-1+b} {}_2F_1(1, 1-a, 2-a-b, 2M) + 2^{1-a-b} N \binom{-2+a+b}{-1+a} {}_2F_1(1, 1-b, 2-a-b, 2N).$$

Here ${}_2F_1$ denotes the hypergeometric function[9]. It shows that the distribution is in some sense linked to the hypergeometric distribution, indicated by the ${}_2F_1$ terms. Even though that function is quite complex, taking derivatives in M or N respectively is straightforward, i.e., $\frac{\partial}{\partial_x} {}_2F_1(p_1,p_2,p_3;x) = \frac{p_1 p_2}{c} {}_2F_1(p_1+1, p_2+1, p_3+1; M)$. Thus, extracting many properties of interest can still be computed

[8] Sometimes also called negative binomial distribution.

[9] More about this closed form and algorithms to compute closed forms alike can be found in [Petkovsek et al. 1996].

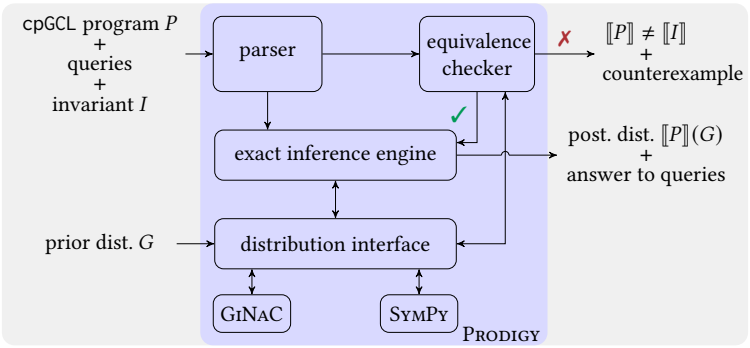


Fig. 4. A sketch of the Prodigy workflow.

exactly using the closed-form expression. It is unknown (to us) whether some loop-free cReDiP invariant program generates this closed-form distribution. However, the GF semantics enables us to prove that the precise semantics of Prog. 13 is captured by checking $\forall a, b \in \mathbb{N}.\ (a, b) \neq (0, 0) \implies I(a, b) = \Phi_{B,P}(I)(a, b)$, combined with the fact that it universally certainly terminates.

6 EMPIRICAL EVALUATION OF PRODIGY

We have implemented our approach in Python as an extension to Prodigy[10][Chen et al. 2022a] – Probability Distributions via GeneratingfunctionologY. The current implementation consists of about 6,000 LOC. The two new features are the implementation of the observe semantics and normalization, as well as a parameter-synthesis approach for finding suitable parameters of distributions to satisfy the invariant condition.

6.1 Implementation of Prodigy

Prodigy implements exact inference for cpGCL programs; its high-level structure is depicted in Figure 4. Given a cpGCL program P (optionally with queries to the output distribution, e.g., expected values, tail bounds and moments) together with a prior distribution G, Prodigy parses the program, performs PGF-based distribution transformations (via the inference engine), and finally outputs the posterior distribution $[\![P]\!](G)$ (plus answers to the queries, if any). For the distribution transformation, Prodigy implements an internal interface acting as an abstract datatype for probability distributions in the form of formal power series. Such an abstraction allows for an easy integration of alternative distribution representations (not necessarily related to generating functions) and various computer algebra systems (CAS) in the backend. Prodigy currently supports SymPy [Meurer et al. 2017] and GiNaC [Bauer et al. 2002; Vollinga 2006]. When (UAST) loops $L = \texttt{while}\,(B)\,\{P'\}$ are encountered, Prodigy asks for a user-provided invariant I and then performs the equivalence check such that it can either infer the output distribution or conclude that $[\![L]\!] \neq [\![I]\!]$ while providing counterexamples σ such that $\Phi_{B,P'}([\![I]\!])(\sigma) \neq [\![I]\!](\sigma)$. In the absence of an invariant, Prodigy is capable of computing under-approximations of the posterior distribution by unfolding the loop up to a specified accuracy or number of loop unrollings.

6.2 Benchmarks

We collected a set of 37 benchmarks, 16 of them related to inferring distributions for loopy programs. This set consists of examples provided by λ-PSI [Gehr et al. 2020], Genfer [Zaiser et al. 2023], and Prodigy. All experiments were evaluated on MacOS Sonoma 14.0 with a 2,4 GHz Quad-Core Intel

[10] https://github.com/LKlinke/Prodigy

Table 4. Benchmarks of loop-free programs; timings are in seconds.

Program	∞	p	Prodigy		λPSI		Genfer
			SymPy	GiNaC	symbolic	dp	
burgler_alarm			1.988	0.012	0.055	0.008	**0.002**
caesar		•	8.377	**0.025**	1.152	0.051	—
digitRecognition			Err.[13]	34.685	96.283	2.818	**0.137**
dnd_handicap			7.760	0.032	0.094	0.039	**0.006**
evidence1			0.348	0.002	0.011	0.002	**<0.001**
evidence2			0.413	0.003	0.014	0.002	**0.001**
function			0.338	0.002	0.001	**<0.001**	0.003
fuzzy_or			67.048	0.227	8.779	4.797	**0.025**
grass			6.706	0.021	0.481	0.089	**0.006**
infer_geom_mix	•		13.723	0.031	0.199	**0.003**	0.139
lin_regression_unbiased			6.700	**0.014**	0.056	0.016	0.918
lucky_throw			Err.[14]	1.560	TO	1.565	**0.455**
max			0.618	0.005	0.020	0.003	**0.001**
monty_hall			2.927	0.033	0.063	**0.004**	0.006
monty_hall_nested			15.694	0.140	0.525	0.017	**0.025**
murder_mystery		•	0.615	0.004	0.020	**0.003**	—
pi			90.931	**0.094**	TO	0.103	—
piranha			0.379	0.003	0.011	0.002	**<0.001**
telephone_operator	•		1.249	0.006	0.058*	Err.[15]	0.006
telephone_operator_param	•	•	5.880	**0.017**	0.108*	0.007	—
twocoins			0.493	0.004	0.011	0.002	**<0.001**

Core i5 and 16GB RAM. For each benchmark, we run Prodigy with both CAS backends, i.e., SymPy and GiNaC. For loop-free benchmarks, Prodigy is compared against λ-PSI[11] and Genfer[12] – the two closest tools (among those in Section 8). As Prodigy is an exact inference engine, all tools are run using *exact arithmetic*. The initial prior distribution is 1 which means all variables are initialized to 0 with probability 1 and no `observe`-violations have occurred. All timings are averaged over 20 iterations per benchmark and we measured the time used for performing inference (computing the posterior distribution). The experiments aim to answer questions in terms of (1) *Effectiveness:* Can Prodigy effectively do exact inference on the selected benchmarks, including equivalence checking and invariant synthesis for programs with loops? (2) *Efficiency:* How does Prodigy compare to the most related tools? How do the CAS backends SymPy and GiNaC compare to each other?

6.3 Experimental Results

General observations. Tables 4 and 5 summarize our experimental results. Our approach is capable of computing posterior distributions for a variety of programs in less than 0.1 seconds. For loop-free benchmarks, *exact* Bayesian inference based on generating functions (Genfer, Prodigy) performs better than λ-PSI on *discrete* probabilistic programs with Genfer being the fastest in most instances. Regarding the timings for Prodigy only, the GiNaC backend is generally about two orders of magnitude faster. Prodigy is the only tool that is able to deal with unbounded loopy programs.

[11]We used the commit `9db68ba9581b7a1211f1514e44e7927af24bd398`.
[12]We used the commit `5911de13f16bc3c28703f1631c5c4847f9ebac9a`.
[13]Exceeding SymPy internal limits for parsing.

Results for loop-free programs. Whereas our focus is on programs featuring unbounded loops, we compared Prodigy to λ-PSI and Genfer for loop-free benchmarks. Table 4 lists the results. The column Program lists the benchmarks. The next column (∞) marks the occurrence of samplings from infinite-support distributions in the benchmark. Column p indicates the presence of symbolic parameters. Finally, columns SymPy, GiNaC, symbolic, dp and genfer list run-times in seconds for the individual backends of Prodigy, λ-PSI, and the tool genfer respectively. Here, dp represents the dynamic programming backend of λ-PSI invoked by using the option `--dp`, and Genfer using exact arithmetic (`--rational`). The timing in boldface marks the fastest variant. The acronym TO stands for time-out, i.e., did not terminate within the time limit of 90 seconds. Entries consisting of "—" indicate the lack of support for this benchmark instance. Timings marked with * refer to results by λ-PSI which contain integral expressions that we like to avoid, however λ-PSI is still able to compute all moments exactly.

Our experiments show that Genfer can be up to two orders of magnitude faster. We emphasize that PSI and Genfer are symbolic engines tailored to solving loop-free inference tasks. Despite this, it turns out that we oftentimes are on par. For the `digitRecognition` example (the most prominent outlier), the speedup of Genfer mostly originates from an optimization in computing the observe-violation probabilities. For loop-free programs, where termination is inherent by design, the necessity to precisely track `observe`-violation probabilities is avoided.Consequently, the observation-violation probability can be computed as the "missing" probability mass in the final distribution. While this methodology is effective in loop-free scenarios, it does not apply to loopy programs and hence was not implemented in Prodigy.

Zaiser and Ong [2023] see automatic differentiation as the key ingredient enabling the fast results of Genfer. Automatic differentiation in the sense of computing n-th derivatives at specific points is done by both Genfer and Prodigy. Whereas Zaiser and Ong [2023] employ a custom implementation, we rely on well-established implementations from SymPy and GiNaC. In fact, the actual differentiation implementation can be exchanged freely. Prodigy's support for loops and parameter synthesis seamlessly integrate with any differentiation method while maintaining the functionality and capitalizing on potential speed enhancements. Moreover, Genfer is unable to deal with non-linear observations as in the `pi` benchmark. The same holds for instances with symbolic parameters. Prodigy outperforms the symbolic engine of λ-PSI on almost every instance whilst Prodigy has a comparable performance to the dynamic programming strategy of λ-PSI.

Results for loopy programs. Table 5 depicts the empirical results for loopy programs. The column Program lists the benchmarks. The columns SymPy and GiNaC report their run-times in seconds when used as backend of Prodigy. The timing in boldface marks the fastest variant. As these benchmarks all include loops, they are not supported by λ-PSI and Genfer.

Recall that reasoning about loops involves an equivalence check against a user-specified invariant program. Finding the right invariant (if it exists in the loop-free `cReDiP` fragment) is intricate. We support the user in discovering such invariants by allowing symbolic parameters for distributions, e.g., one can write `geom`(p) where p is a symbolic parameter. For benchmarks subject to parameter synthesis, we also provide the anticipated parameter constraints (or values) inferred automatically by Prodigy. Whenever this is the case, we point out that for the GiNaC timings, discharging the resulting equation systems is achieved using SymPy solvers, which is due to the missing functionality of GiNaC to solve these equation systems. Overall, GiNaC is faster than SymPy by about two orders of magnitude, as is similar to the loop-free benchmarks.

[14]Reached maximum recursion limit

[15]The `--dp` strategy produces $p(x, d) = 0$ which is an incorrect result.

Table 5. Exact inference results for loopy probabilistic programs (those with parameter synthesis are marked by _param); timings are given in seconds.

Program	SymPy		GiNaC	
dep_bern	13.354		**0.457**	
endless_conditioning	1.148		**0.012**	
geometric	3.757		**0.031**	
ky_die	21.562		**0.209**	
n_geometric	3.050		**0.038**	
random_walk	3.439		**0.047**	
trivial_iid	6.444		**0.075**	
bit_flip_conditioning	31.030		**0.322**	
dueling_cowboys_param	6.147	for any p, q	**0.065**	for any p, q
geometric_param	4.888	$p = \frac{1}{3}$	**0.262**	$p = \frac{1}{3}$
ky_die_param	36.619	$p = \frac{2}{3}, q = \frac{1}{2}$	**1.298**	$p = \frac{2}{3}, q = \frac{1}{2}$
negative_binomial_param	2.814	for any p	**0.047**	for any p
n_geometric_param	5.365	$p = \frac{q}{3}$	**0.133**	$p = \frac{q}{3}$
random_walk_param	5.114	$p = \frac{1}{2}$	**0.274**	$p = \frac{1}{2}$
bit_flip_cond_param	58.599	$p = \frac{13}{28}, q = \frac{3}{7}, r = \frac{2}{7}$	**0.887**	$p = \frac{13}{28}, q = \frac{3}{7}, r = \frac{2}{7}$
brp_obs_param	TO		**77.732**	$p = 10^{-10}$

It is also worth noting that Prodigy is potentially applicable to practical randomized algorithms beyond toy programs like random walks. These applications include loop-free benchmarks such as `digitRecognition` for recognizing written digits based on observed data samples, as well as the unbounded loopy program modeling the bounded retransmission protocol (`brp_obs_param`):

Example 26 (Bounded Retransmission Protocol). Prog. 14 describes a conditioned variant of the bounded retransmission protocol (BRP) [Batz et al. 2023; D'Argenio et al. 2001] which attempts to transmit s packets over a lossy channel. Each individual packet gets lost with probability 1%. The transmission is successful, if *no packet* needs more than 4 resends. Further, we observe that all but the last 9 packets are received successfully without any additional resends. Fig. 5 illustrates the protocol as a Markov chain. Note that the number of packets to be sent is parametrized by the (possibly infinite-support) initial distribution of s and f, modeling an *infinite family* of finite-state Markov chains. This renders techniques like probabilistic model checking [Katoen 2016] infeasible.

Provided with a suitable invariant (cf. [Klinkenberg et al. 2024a, Appx. E]) with parameter p in the probabilities, Prodigy infers that, with $p = 10^{-10}$, Prog. 14 is equivalent to this invariant, thereby yielding the exact output distribution (for any initial distribution of s with rational closed form) in the form of a PGF. From this PGF, we can derive, e.g., with input $s \sim \texttt{geom}(1/2)$, the *transmission-failure probability* of BRP, i.e., the probability that Prog. 14 terminates with $f > 4$ is around 9.9789×10^{-11} (see [Klinkenberg et al. 2024a, Appx. E]).

From a syntactic point of view, the BRP may seem intricate. Yet semantically, it represents the structure of the original program's underlying Markov chain (Fig. 5) in a straightforward manner. For all but the last 9 packets, no transmission attempt is allowed to fail. If starting with at most 9 packets to send in total, the initial state might already indicate some failed attempts for the first packet to transmit. In this case, the first packet sent has less than 5 retries to successfully complete the transmission. Afterwards, for each of the remaining packets, transmission either fails with some probability p or is successful and continues with the next packet.

```
while ( s > 0 ∧ f ≤ 4 ) {
  /* packet loss */
  {observe ( s ≤ 9 ) ; f := f + 1}
  [1/100]
  /* packet received */
  {f := 0 ; s := s − 1}
}
```

Prog. 14. A conditioned variant of BRP.

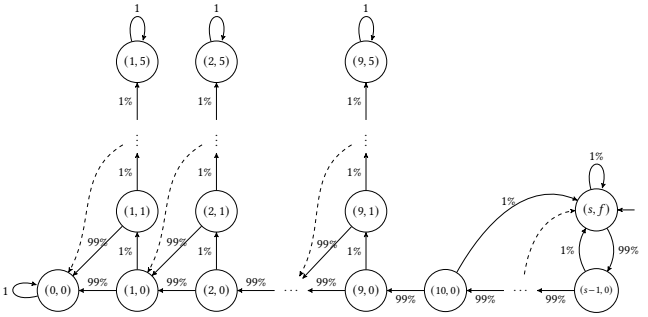


Fig. 5. The Markov chain illustrating Prog. 14.

7 LIMITATIONS OF EXACT INFERENCE USING EFPS

We discuss some limitations of the presented inference approach considering guard evaluations, non-rational probabilities and scalability. Prog. 16 models a variant of the famous Collatz algorithm [Andrei and Masalagiu 1998]. The Collatz conjecture states that for all positive integers m there exists $n \in \mathbb{N}$ such that for the Collatz function $C(m) := {}^{n}/_{2}$ for $n \equiv (0 \bmod 2)$ and $3n+1$ otherwise; the n-th fold iteration of the function is $C^n(m) = 1$. We have adapted the program syntax slightly and make use of the `loop` statement to represent the n-fold repetition of a code block. The program basically behaves as the usual Collatz function with the only exception that in the case where a number is divisible by two, we have a small chance not dividing x by 2 but instead executing the `else` branch. Note that the instruction $x \equiv_2 0 (\bmod 2)$ still preserves rational closed forms as we can compute its semantics by $\frac{F(\mathrm{X})+F(-\mathrm{X})}{2}$. When analyzing the run-times of our tool on this program we observe surprising results: for $(n = 1)$ we obtain a result in 0.010631 seconds; $(n = 2)$ is computed in 0.049891 seconds and for $(n = 3)$ it suddenly increases to 88.689832 seconds. We think that this phenomenon arises from the fact that evaluating expressions like $x \equiv 0 \pmod 2$ repeatedly, gets increasingly difficult as it is implemented in Prodigy by means of arithmetic progressions.

Another challenge is guard evaluation, i.e., filtering out the corresponding terms of a formal power series such as $\langle F \rangle_B$ for `if`-statements. In case we are interested in the relation between two variables (like $x = y$) when both have marginal distributions with infinite support, Prodigy cannot compute the result. As an approximation heuristic it computes under-approximations of the *exact* posterior distribution. Note that if either x or y has a finite-support marginal distribution, the posterior is computed by enumeration. An interesting example why one cannot even strive for such a potential closed-form operation preserving rational closed forms is Prog. 15. For this program, its variable r evaluates to 1 with *non-rational*, not even algebraic probability ${}^{1}/_{\pi}$ after termination [Flajolet et al. 2011] – thus beyond `cReDiP` capabilities. An interesting open question is to determine what syntactic restrictions *exactly capture rational* closed forms.

As a final observation we emphasize that Prodigy's performance is proportional to the size of constants in the programs. Assume, e.g., a guard $x > n$, where n is a constant. For larger n, the closed-form operation of computing the n-th formal derivative takes an increasing amount of time.

8 RELATED WORK

We review a non-exhaustive list of related work in probabilistic inference, ranging from invariant-based verification techniques to inference techniques based on sampling and symbolic methods.

Invariant-based verification. As a means to avoid intractable fixed point computations, the correctness of loopy probabilistic programs can often be established by inferring specific (inductive)

```
x ≔ geom(1/4) ⨟
y ≔ geom(1/4) ⨟
t ≔ x + y ⨟
{ t ≔ t + 1 } [5/9] { skip } ⨟
r ≔ 1 ⨟
loop(3){
  s ≔ iid(bernoulli(1/2), 2t) ⨟
  if ( s ≠ t ) {r ≔ 0}
}
```

Prog. 15. Non-algebraic probabilities.

```
x ≔ geom(1/2) ⨟
loop(n){
  if ( x ≡ 0 (mod 2) ) {
    { x ≔ 3 ∗ x + 1 } [1/10]
      { x ≔ 1/2 ∗ x }
  } else {
    x ≔ 3 ∗ x + 1
  }
}
```

Prog. 16. Probabilistic Collatz program.

bounds on expectations, called *quantitative loop invariants* [McIver and Morgan 2005]. There are a variety of results on synthesizing quantitative invariants, including (semi-)automated techniques based on *martingales* [Barthe et al. 2016; Chakarov and Sankaranarayanan 2013, 2014; Chatterjee et al. 2020, 2017; Takisaka et al. 2021], *recurrence solving* [Bartocci et al. 2019, 2020b], *invariant learning* [Bao et al. 2022], and *constraint solving* [Chen et al. 2015; Feng et al. 2017; Gretz et al. 2013; Katoen et al. 2010], particularly via *satisfiability modulo theories* (SMT) [Batz et al. 2023, 2021, 2020].

Alternative state-of-the-art verification approaches include *bounded model checking* [Jansen et al. 2016] for verifying probabilistic programs with nondeterminism and conditioning as well as various forms of *value iteration* [Baier et al. 2017; Hartmanns and Kaminski 2020; Quatmann and Katoen 2018] for determining reachability probabilities in finite Markov models.

Sampling-based inference. Most existing probabilistic programming languages implement *sampling*-based inference algorithms rooted in the principles of Monte Carlo [Metropolis and Ulam 1949], thereby yielding numerical approximations of the exact results, see, e.g., [Gram-Hansen 2021]. Such languages include Anglican [Wood et al. 2014], BLOG [Milch et al. 2005], BUGS [Spiegelhalter et al. 1995], Infer.NET [Minka et al. 2018], R2 [Nori et al. 2014], Stan [Stan Development Team 2022], etc. In contrast, we are concerned with inference techniques that produce *exact* results.

Symbolic inference. In response to the aforementioned challenges (i) and (ii) in exact probabilistic inference, Klinkenberg et al. [2020] proposed a program semantics based on *probability generating functions*. This PGF-based semantics allows for exact quantitative reasoning for, e.g., deciding probabilistic equivalence [Chen et al. 2022a] and proving non-almost-sure termination [Klinkenberg et al. 2020] for certain probabilistic programs *without conditioning*.

Extensions of PGF-based approaches to programs with conditioning have been initiated in [Klinkenberg et al. 2023; Zaiser et al. 2023]; the latter suggested the use of automatic differentiation in the evaluation of PGFs, but the paper addresses *loop-free programs only*. Combining conditioning and possibly non-terminating behaviors (introduced through loops) substantially complicates the computation of final probability distributions and normalization constants. Another difference is that Zaiser et al. provide truncated posterior distributions together with the first four centralized moments. We, in contrast, develop a symbolic representation of the full posterior distribution.

As an alternative to PGFs, many probabilistic systems employ *probability density function* (PDF) representations of distributions, e.g., (λ)PSI [Gehr et al. 2016, 2020], AQUA [Huang et al. 2021] and Hakaru [Narayanan et al. 2016], as well as the density compiler in [Bhat et al. 2012, 2017]. These systems are dedicated to inference for programs encoding joint (discrete-)continuous distributions

with conditioning. Reasoning about the underlying PDF representations, however, amounts to resolving complex integral expressions in order to answer inference queries. Furthermore, (λ)PSI admits *only bounded looping behaviors*. Dice [Holtzen et al. 2020] employs weighted model counting to enable potentially scalable exact inference for discrete probabilistic programs, yet is also confined to statically bounded loops. Stein and Staton [2021] proposed a denotational semantics based on Markov categories for continuous probabilistic programs with exact conditioning and bounded looping behaviors. A similar direction is taken by Bichsel et al. [2018]. They investigate the connections between `observe`-violations, non-termination, and errors raised by, e.g., division by zero; their semantics is based on Markov kernels. A recently proposed language PERPL [Chiang et al. 2023] compiles probabilistic programs with unbounded recursion into systems of polynomial equations and solves them directly for least fixed points using numerical methods. A related approach by Stuhlmüller and Goodman [2012] uses dynamic programming techniques transforming probabilistic programs with unbounded recursion into factored sum-product networks, i.e., a particular way of representing an equation system. However, this technique cannot handle infinite-support distributions. The tool Mora [Bartocci et al. 2020a,b] supports exact inference for various types of Bayesian networks, but relies on a restricted form of intermediate representation known as prob-solvable loops, whose behaviors can be expressed by a system of C-finite recurrences admitting closed-form solutions.

Finally, we refer interested readers to [Sheldon et al. 2018; Winner and Sheldon 2016; Winner et al. 2017] for a related line of research from the machine learning community, which exploits PGF-based exact inference – not for probabilistic programs – but for dedicated types of graphical models with latent count variables.

9 CONCLUSION

We have presented an exact Bayesian inference approach for probabilistic programs with (possibly unbounded) loops and conditioning. The core of this approach is a denotational semantics that symbolically encodes distributions as probability generating functions. We showed how our PGF-based exact inference facilitates (semi-)automated inference, equivalence checking, and invariant synthesis of probabilistic programs. Our implementation in Prodigy shows promise: It can do exact inference for various infinite-state loopy programs and exhibits comparable performance to state-of-the-art exact inference tools over loop-free benchmarks.

The possibility to incorporate symbolic parameters in GF representations can enable the application of well-established optimization methods, e.g., maximum-likelihood estimations and parameter fitting, to probabilistic inference. Characterizing the family of programs and invariants which admit a potentially complete eSOP-based synthesis approach would be of particular interest. Additionally, future research directions include extending exact inference to continuous distributions by utilizing characteristic functions as the continuous counterpart to PGFs. Furthermore, there is an intriguing connection to be explored between quantitative reasoning about loops and the positivity problem of recurrence sequences [Ouaknine and Worrell 2014], which is induced by loop unfolding.

ACKNOWLEDGMENTS

Lutz Klinkenberg and Joost-Pieter Katoen are supported by ERC AdG Grant 787914; Darion Haase is supported by the DFG RTG 2236 UnRAVeL; Mingshuai Chen is supported by the ZJNSF Major Program under grant No. LD24F020013 and by the ZJU Education Foundation's Qizhen Talent program. The authors would like to thank the anonymous reviewers for their constructive feedback on this article and Leo Mommers for his assistance in producing the benchmark results and his work on part of the implementation.

DATA-AVAILABILITY STATEMENT

The software that supports Section 6 is available on Zenodo [Klinkenberg et al. 2024b].

REFERENCES

Nathanael L. Ackerman, Cameron E. Freer, and Daniel M. Roy. 2019. On the Computability of Conditional Probability. *J. ACM* 66, 3 (2019). https://doi.org/10.1145/3321699

Ştefan Andrei and Cristian Masalagiu. 1998. About the Collatz conjecture. *Acta Informatica* 35, 2 (1998), 167–179. https://doi.org/10.1007/S002360050117

Christel Baier, Joachim Klein, Linda Leuschner, David Parker, and Sascha Wunderlich. 2017. Ensuring the Reliability of Your Model Checker: Interval Iteration for Markov Decision Processes. In *CAV (2) (LNCS, Vol. 10426)*. Springer, 160–180. https://doi.org/10.1007/978-3-319-63387-9_8

Jialu Bao, Nitesh Trivedi, Drashti Pathak, Justin Hsu, and Subhajit Roy. 2022. Data-Driven Invariant Learning for Probabilistic Programs. In *CAV (1) (LNCS, Vol. 13371)*. Springer, 33–54. https://doi.org/10.1007/978-3-031-13185-1_3

Gilles Barthe, Thomas Espitau, Luis María Ferrer Fioriti, and Justin Hsu. 2016. Synthesizing Probabilistic Invariants via Doob's Decomposition. In *CAV (1) (LNCS, Vol. 9779)*. Springer, 43–61. https://doi.org/10.1007/978-3-319-41528-4_3

Gilles Barthe, Joost-Pieter Katoen, and Alexandra Silva (Eds.). 2020. *Foundations of Probabilistic Programming*. Cambridge University Press. https://doi.org/10.1017/9781108770750

Ezio Bartocci, Laura Kovács, and Miroslav Stankovic. 2019. Automatic Generation of Moment-Based Invariants for Prob-Solvable Loops. In *ATVA (LNCS, Vol. 11781)*. Springer, 255–276. https://doi.org/10.1007/978-3-030-31784-3_15

Ezio Bartocci, Laura Kovács, and Miroslav Stankovic. 2020a. Analysis of Bayesian Networks via Prob-Solvable Loops. In *ICTAC (LNCS, Vol. 12545)*. Springer, 221–241. https://doi.org/10.1007/978-3-030-64276-1_12

Ezio Bartocci, Laura Kovács, and Miroslav Stankovic. 2020b. Mora - Automatic Generation of Moment-Based Invariants. In *TACAS (1) (LNCS, Vol. 12078)*. Springer, 492–498. https://doi.org/10.1007/978-3-030-45190-5_28

Kevin Batz, Mingshuai Chen, Sebastian Junges, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. 2023. Probabilistic Program Verification via Inductive Synthesis of Inductive Invariants. In *TACAS (2) (LNCS, Vol. 13994)*. Springer, 410–429. https://doi.org/10.1007/978-3-031-30820-8_25

Kevin Batz, Mingshuai Chen, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Philipp Schröer. 2021. Latticed k-Induction with an Application to Probabilistic Programs. In *CAV (2) (LNCS, Vol. 12760)*. Springer, 524–549. https://doi.org/10.1007/978-3-030-81688-9_25

Kevin Batz, Sebastian Junges, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Philipp Schröer. 2020. PrIC3: Property Directed Reachability for MDPs. In *CAV (2) (LNCS, Vol. 12225)*. Springer, 512–538. https://doi.org/10.1007/978-3-030-53291-8_27

Christian Bauer, Alexander Frink, and Richard Kreckel. 2002. Introduction to the GiNaC Framework for Symbolic Computation within the C++ Programming Language. *J. Symb. Comput.* 33, 1 (2002), 1–12. https://doi.org/10.1006/jsco.2001.0494

Sooraj Bhat, Ashish Agarwal, Richard W. Vuduc, and Alexander G. Gray. 2012. A Type Theory for Probability Density Functions. In *POPL*. ACM, 545–556. https://doi.org/10.1145/2103656.2103721

Sooraj Bhat, Johannes Borgström, Andrew D. Gordon, and Claudio V. Russo. 2017. Deriving Probability Density Functions from Probabilistic Functional Programs. *Log. Methods Comput. Sci.* 13, 2 (2017). https://doi.org/10.23638/LMCS-13(2:16)2017

Benjamin Bichsel, Timon Gehr, and Martin T. Vechev. 2018. Fine-Grained Semantics for Probabilistic Programs. In *ESOP (LNCS, Vol. 10801)*. Springer, 145–185. https://doi.org/10.1007/978-3-319-89884-1_6

Olivier Bournez and Florent Garnier. 2005. Proving Positive Almost-Sure Termination. In *RTA (LNCS, Vol. 3467)*. Springer, 323–337. https://doi.org/10.1007/978-3-540-32033-3_24

Michael Carbin, Sasa Misailovic, and Martin C. Rinard. 2016. Verifying quantitative reliability for programs that execute on unreliable hardware. *Commun. ACM* 59, 8 (2016), 83–91. https://doi.org/10.1145/2958738

Milan Češka, Christian Dehnert, Nils Jansen, Sebastian Junges, and Joost-Pieter Katoen. 2019. Model Repair Revamped – On the Automated Synthesis of Markov Chains. In *From Reactive Systems to Cyber-Physical Systems (LNCS, Vol. 11500)*. Springer, 107–125. https://doi.org/10.1007/978-3-030-31514-6_7

Aleksandar Chakarov and Sriram Sankaranarayanan. 2013. Probabilistic Program Analysis with Martingales. In *CAV (LNCS, Vol. 8044)*. Springer, 511–526. https://doi.org/10.1007/978-3-642-39799-8_34

Aleksandar Chakarov and Sriram Sankaranarayanan. 2014. Expectation Invariants for Probabilistic Program Loops as Fixed Points. In *SAS (LNCS, Vol. 8723)*. Springer, 85–100. https://doi.org/10.1007/978-3-319-10936-7_6

Krishnendu Chatterjee, Hongfei Fu, and Amir Kafshdar Goharshady. 2016. Termination Analysis of Probabilistic Programs Through Positivstellensatz's. In *CAV (1) (LNCS, Vol. 9779)*. Springer, 3–22. https://doi.org/10.1007/978-3-319-41528-4_1

Krishnendu Chatterjee, Hongfei Fu, and Petr Novotný. 2020. Termination Analysis of Probabilistic Programs with Martingales. In *Foundations of Probabilistic Programming*, Gilles Barthe, Joost-Pieter Katoen, and Alexandra Silva (Eds.). Cambridge

University Press, 221–258. https://doi.org/10.1017/9781108770750.008

Krishnendu Chatterjee, Petr Novotný, and Dorde Zikelic. 2017. Stochastic Invariants for Probabilistic Termination. In *POPL*. ACM, 145–160. https://doi.org/10.1145/3009837.3009873

Mingshuai Chen, Joost-Pieter Katoen, Lutz Klinkenberg, and Tobias Winkler. 2022a. Does a Program Yield the Right Distribution? Verifying Probabilistic Programs via Generating Functions. In *CAV (1) (LNCS, Vol. 13371)*. Springer, 79–101. https://doi.org/10.1007/978-3-031-13185-1_5

Mingshuai Chen, Joost-Pieter Katoen, Lutz Klinkenberg, and Tobias Winkler. 2022b. Does a Program Yield the Right Distribution? Verifying Probabilistic Programs via Generating Functions. *CoRR* abs/2205.01449 (2022). https://doi.org/10.48550/ARXIV.2205.01449

Yu-Fang Chen, Chih-Duo Hong, Bow-Yaw Wang, and Lijun Zhang. 2015. Counterexample-Guided Polynomial Loop Invariant Generation by Lagrange Interpolation. In *CAV (1) (LNCS, Vol. 9206)*. Springer, 658–674. https://doi.org/10.1007/978-3-319-21690-4_44

David Chiang, Colin McDonald, and Chung-chieh Shan. 2023. Exact Recursive Probabilistic Programming. *Proc. ACM Program. Lang.* 7, OOPSLA1 (2023), 665–695. https://doi.org/10.1145/3586050

Gregory F. Cooper. 1990. The Computational Complexity of Probabilistic Inference Using Bayesian Belief Networks. *Artif. Intell.* 42, 2-3 (1990), 393–405. https://doi.org/10.1016/0004-3702(90)90060-D

Fredrik Dahlqvist, Alexandra Silva, and Dexter Kozen. 2020. Semantics of Probabilistic Programming: A Gentle Introduction. In *Foundations of Probabilistic Programming*, Gilles Barthe, Joost-Pieter Katoen, and Alexandra Silva (Eds.). Cambridge University Press, 1–42. https://doi.org/10.1017/9781108770750.002

Pedro R. D'Argenio, Bertrand Jeannet, Henrik Ejersbo Jensen, and Kim Guldstrand Larsen. 2001. Reachability Analysis of Probabilistic Systems by Successive Refinements. In *PAPM-PROBMIV (Lecture Notes in Computer Science, Vol. 2165)*. Springer, 39–56. https://doi.org/10.1007/3-540-44804-7_3

Devdatt P Dubhashi and Alessandro Panconesi. 2009. *Concentration of Measure for the Analysis of Randomized Algorithms*. Cambridge University Press.

Shenghua Feng, Mingshuai Chen, Han Su, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Naijun Zhan. 2023. Lower Bounds for Possibly Divergent Probabilistic Programs. *Proc. ACM Program. Lang.* 7, OOPSLA1 (2023), 696–726. https://doi.org/10.1145/3586051

Yijun Feng, Lijun Zhang, David N. Jansen, Naijun Zhan, and Bican Xia. 2017. Finding Polynomial Loop Invariants for Probabilistic Programs. In *ATVA (LNCS, Vol. 10482)*. Springer, 400–416. https://doi.org/10.1007/978-3-319-68167-2_26

Philippe Flajolet, Maryse Pelletier, and Michèle Soria. 2011. On Buffon Machines and Numbers. In *SODA*. SIAM, 172–183. https://doi.org/10.1137/1.9781611973082.15

Philippe Flajolet and Robert Sedgewick. 2009. *Analytic Combinatorics*. Cambridge University Press. https://doi.org/10.1017/CBO9780511801655

Daniel J. Fremont, Edward Kim, Tommaso Dreossi, Shromona Ghosh, Xiangyu Yue, Alberto L. Sangiovanni-Vincentelli, and Sanjit A. Seshia. 2023. Scenic: a language for scenario specification and data generation. *Mach. Learn.* 112, 10 (2023), 3805–3849. https://doi.org/10.1007/S10994-021-06120-5

Timon Gehr, Sasa Misailovic, and Martin T. Vechev. 2016. PSI: Exact Symbolic Inference for Probabilistic Programs. In *CAV (1) (LNCS, Vol. 9779)*. Springer, 62–83. https://doi.org/10.1007/978-3-319-41528-4_4

Timon Gehr, Samuel Steffen, and Martin T. Vechev. 2020. λPSI: Exact Inference for Higher-Order Probabilistic Programs. In *PLDI*. ACM, 883–897. https://doi.org/10.1145/3385412.3386006

Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. 2014. Probabilistic Programming. In *FOSE*. ACM, 167–181. https://doi.org/10.1145/2593882.2593900

Bradley Gram-Hansen. 2021. *Extending probabilistic programming systems and applying them to real-world simulators*. Ph. D. Dissertation. University of Oxford.

Friedrich Gretz, Joost-Pieter Katoen, and Annabelle McIver. 2013. Prinsys - On a Quest for Probabilistic Loop Invariants. In *QEST (LNCS, Vol. 8054)*. Springer, 193–208. https://doi.org/10.1007/978-3-642-40196-1_17

Marcel Hark, Benjamin Lucien Kaminski, Jürgen Giesl, and Joost-Pieter Katoen. 2020. Aiming low is harder: Induction for lower bounds in probabilistic program verification. *Proc. ACM Program. Lang.* 4, POPL (2020), 37:1–37:28. https://doi.org/10.1145/3371105

Arnd Hartmanns and Benjamin Lucien Kaminski. 2020. Optimistic Value Iteration. In *CAV (2) (LNCS, Vol. 12225)*. Springer, 488–511. https://doi.org/10.1007/978-3-030-53291-8_26

Steven Holtzen, Guy Van den Broeck, and Todd D. Millstein. 2020. Scaling Exact Inference for Discrete Probabilistic Programs. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 140:1–140:31. https://doi.org/10.1145/3428208

Zixin Huang, Saikat Dutta, and Sasa Misailovic. 2021. AQUA: Automated Quantized Inference for Probabilistic Programs. In *ATVA (LNCS, Vol. 12971)*. Springer, 229–246. https://doi.org/10.1007/978-3-030-88885-5_16

Wolfram Research, Inc. 2023. Mathematica, Version 13.3. https://www.wolfram.com/mathematica Champaign, IL, 2023.

Jules Jacobs. 2021. Paradoxes of probabilistic programming: And how to condition on events of measure zero with infinitesimal probabilities. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–26. https://doi.org/10.1145/3434339

Nils Jansen, Christian Dehnert, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Lukas Westhofen. 2016. Bounded Model Checking for Probabilistic Programs. In *ATVA (LNCS, Vol. 9938)*. 68–85. https://doi.org/10.1007/978-3-319-46520-3_5

Norman L Johnson, Adrienne W Kemp, and Samuel Kotz. 2005. *Univariate Discrete Distributions*. Vol. 444. John Wiley & Sons. https://doi.org/10.1002/0471715816

Benjamin Lucien Kaminski. 2019. *Advanced weakest precondition calculi for probabilistic programs*. Ph. D. Dissertation. RWTH Aachen University. https://doi.org/10.18154/RWTH-2019-01829

Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. 2019. On the Hardness of Analyzing Probabilistic Programs. *Acta Inform.* 56, 3 (2019), 255–285. https://doi.org/10.1007/s00236-018-0321-1

Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. 2018. Weakest Precondition Reasoning for Expected Runtimes of Randomized Algorithms. *J. ACM* 65, 5 (2018), 30:1–30:68. https://doi.org/10.1145/3208102

Joost-Pieter Katoen. 2016. The Probabilistic Model Checking Landscape. In *LICS*. ACM, 31–45. https://doi.org/10.1145/2933575.2934574

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 (LNCS, Vol. 6337)*. Springer, 390–406. https://doi.org/10.1007/978-3-642-15769-1_24

Lutz Klinkenberg, Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Joshua Moerman, and Tobias Winkler. 2020. Generating Functions for Probabilistic Programs. In *LOPSTR (LNCS, Vol. 12561)*. Springer, 231–248. https://doi.org/10.1007/978-3-030-68446-4_12

Lutz Klinkenberg, Christian Blumenthal, Mingshuai Chen, Darion Haase, and Joost-Pieter Katoen. 2024a. Exact Bayesian Inference for Loopy Probabilistic Programs using Generating Functions. *CoRR* abs/2307.07314v4 (2024). https://doi.org/10.48550/ARXIV.2307.07314 arXiv:2307.07314v4

Lutz Klinkenberg, Christian Blumenthal, Mingshuai Chen, Darion Haase, and Joost-Pieter Katoen. 2024b. *Exact Bayesian Inference for Loopy Probabilistic Programs using Generating Functions – Artifact*. https://doi.org/10.5281/zenodo.10782412

Lutz Klinkenberg, Mingshuai Chen, Joost-Pieter Katoen, and Tobias Winkler. 2023. Exact Probabilistic Inference Using Generating Functions. *CoRR* abs/2302.00513 (2023). https://doi.org/10.48550/ARXIV.2302.00513

Dexter Kozen. 1981. Semantics of Probabilistic Programs. *J. Comput. Syst. Sci.* 22, 3 (1981), 328–350. https://doi.org/10.1016/0022-0000(81)90036-2

Johan Henri Petrus Kwisthout. 2009. *The computational complexity of probabilistic networks*. Ph. D. Dissertation. Utrecht University.

Michael L. Littman, Judy Goldsmith, and Martin Mundhenk. 1998. The Computational Complexity of Probabilistic Planning. *J. Artif. Intell. Res.* 9 (1998), 1–36. https://doi.org/10.1613/JAIR.505

Annabelle McIver and Carroll Morgan. 2005. *Abstraction, Refinement and Proof for Probabilistic Systems*. Springer. https://doi.org/10.1007/B138392

Nicholas Metropolis and Stanisław Ulam. 1949. The Monte Carlo Method. *J. Am. Stat. Assoc.* 44, 247 (1949), 335–341. https://doi.org/10.1080/01621459.1949.10483310

Aaron Meurer et al. 2017. SymPy: Symbolic computing in Python. *PeerJ Comput. Sci.* 3 (2017), e103. https://doi.org/10.7717/peerj-cs.103

Brian Milch, Bhaskara Marthi, Stuart Russell, David A. Sontag, Daniel L. Ong, and Andrey Kolobov. 2005. BLOG: Probabilistic Models with Unknown Objects. In *IJCAI*. 1352–1359.

Tom Minka, John M. Winn, John P. Guiver, Yordan Zaykov, Dany Fabian, and John Bronskill. 2018. Infer.NET 0.3. http://dotnet.github.io/infer Microsoft Research Cambridge.

Michael Mitzenmacher and Eli Upfal. 2005. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press. https://doi.org/10.1017/CBO9780511813603

Marcel Moosbrugger, Miroslav Stankovic, Ezio Bartocci, and Laura Kovács. 2022. This is the moment for probabilistic loops. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 1497–1525. https://doi.org/10.1145/3563341

Praveen Narayanan, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov. 2016. Probabilistic Inference by Program Transformation in Hakaru (System Description). In *FLOPS (LNCS, Vol. 9613)*. Springer, 62–79. https://doi.org/10.1007/978-3-319-29604-3_5

Aditya V. Nori, Chung-Kil Hur, Sriram K. Rajamani, and Selva Samuel. 2014. R2: An Efficient MCMC Sampler for Probabilistic Programs. In *AAAI*. AAAI Press, 2476–2482. https://doi.org/10.1609/AAAI.V28I1.9060

Federico Olmedo, Friedrich Gretz, Nils Jansen, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Annabelle McIver. 2018. Conditioning in Probabilistic Programming. *ACM Trans. Program. Lang. Syst.* 40, 1 (2018), 4:1–4:50. https://doi.org/10.1145/3156018

Joël Ouaknine and James Worrell. 2014. On the Positivity Problem for Simple Linear Recurrence Sequences. In *ICALP (2) (LNCS, Vol. 8573)*. Springer, 318–329. https://doi.org/10.1007/978-3-662-43951-7_27

David Park. 1969. Fixpoint Induction and Proofs of Program Properties. *Machine intelligence* 5 (1969).

Marko Petkovsek, Herbert S Wilf, and Doron Zeilberger. 1996. *A = B*. CRC Press.

Tim Quatmann and Joost-Pieter Katoen. 2018. Sound Value Iteration. In *CAV (1) (LNCS, Vol. 10981)*. Springer, 643–661. https://doi.org/10.1007/978-3-319-96145-3_37

Dan Roth. 1996. On the Hardness of Approximate Reasoning. *Artif. Intell.* 82, 1 (1996), 273–302. https://doi.org/10.1016/0004-3702(94)00092-1

Nasser Saheb-Djahromi. 1978. Probabilistic LCF. In *MFCS (LNCS, Vol. 64)*. Springer, 442–451. https://doi.org/10.1007/3-540-08921-7_92

Daniel Sheldon, Kevin Winner, and Debora Sujono. 2018. Learning in Integer Latent Variable Models with Nested Automatic Differentiation. In *ICML (PMLR, Vol. 80)*. PMLR, 4622–4630.

David J. Spiegelhalter, Andrew Thomas, Nicola G. Best, and Walter R. Gilks. 1995. *BUGS: Bayesian Inference Using Gibbs Sampling, Version 0.50.*

Stan Development Team. 2022. *Stan Modeling Language Users Guide and Reference Manual, Version 2.31.*

Dario Stein and Sam Staton. 2021. Compositional Semantics for Probabilistic Programs with Exact Conditioning. In *LICS*. IEEE, 1–13. https://doi.org/10.1109/LICS52264.2021.9470552

Andreas Stuhlmüller and Noah D. Goodman. 2012. A Dynamic Programming Algorithm for Inference in Recursive Probabilistic Programs. *CoRR* abs/1206.3555 (2012). https://doi.org/10.48550/arXiv.1206.3555

Toru Takisaka, Yuichiro Oyabu, Natsuki Urabe, and Ichiro Hasuo. 2021. Ranking and Repulsing Supermartingales for Reachability in Randomized Programs. *ACM Trans. Program. Lang. Syst.* 43, 2 (2021), 5:1–5:46. https://doi.org/10.1145/3450967

Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. 2018. An Introduction to Probabilistic Programming. *CoRR* abs/1809.10756 (2018). https://doi.org/10.48550/arXiv.1809.10756

Jens Vollinga. 2006. GiNaC–Symbolic Computation with C++. *Nucl. Instrum. Methods Phys. Res.* 559, 1 (2006), 282–284. https://doi.org/10.1016/j.nima.2005.11.155

Di Wang, Jan Hoffmann, and Thomas W. Reps. 2021a. Central moment analysis for cost accumulators in probabilistic programs. In *PLDI*. ACM, 559–573. https://doi.org/10.1145/3453483.3454062

Jinyi Wang, Yican Sun, Hongfei Fu, Krishnendu Chatterjee, and Amir Kafshdar Goharshady. 2021b. Quantitative analysis of assertion violations in probabilistic programs. In *PLDI*. ACM, 1171–1186. https://doi.org/10.1145/3453483.3454102

Herbert S Wilf. 2005. *Generatingfunctionology*. CRC press.

Kevin Winner and Daniel Sheldon. 2016. Probabilistic Inference with Generating Functions for Poisson Latent Variable Models. In *NIPS*. 2640–2648.

Kevin Winner, Debora Sujono, and Daniel Sheldon. 2017. Exact Inference for Integer Latent-Variable Models. In *ICML (PMLR, Vol. 70)*. PMLR, 3761–3770.

Frank D. Wood, Jan-Willem van de Meent, and Vikash Mansinghka. 2014. A New Approach to Probabilistic Programming Inference. In *AISTATS*, Vol. 33. JMLR.org, 1024–1032.

Fabian Zaiser, Andrzej S. Murawski, and C.-H. Luke Ong. 2023. Exact Bayesian Inference on Discrete Models via Probability Generating Functions: A Probabilistic Programming Approach. In *NeurIPS*. To appear.

Fabian Zaiser and C.-H. Luke Ong. 2023. *Exact Inference for Discrete Probabilistic Programs via Generating Functions*. https://popl23.sigplan.org/details/lafi-2023-papers/10/Exact-Inference-for-Discrete-Probabilistic-Programs-via-Generating-Functions

Received 21-OCT-2023; accepted 2024-02-24