Does a Program Yield the Right Distribution?

Verifying Probabilistic Programs via Generating Functions

Mingshuai Chen(✉), Joost-Pieter Katoen(✉), Lutz Klinkenberg(✉), and Tobias Winkler(✉)

RWTH Aachen University, Aachen, Germany
{chenms,katoen,lutz.klinkenberg, tobias.winkler}@cs.rwth-aachen.de

Abstract. We study discrete probabilistic programs with potentially unbounded looping behaviors over an infinite state space. We present, to the best of our knowledge, *the first decidability result for the problem of determining whether such a program generates exactly a specified distribution over its outputs* (provided the program terminates almost-surely). The class of distributions that can be specified in our formalism consists of standard distributions (geometric, uniform, etc.) and finite convolutions thereof. Our method relies on representing these (possibly infinite-support) distributions as *probability generating functions* which admit effective arithmetic operations. We have automated our techniques in a tool called Prodigy, which supports automatic invariance checking, compositional reasoning of nested loops, and efficient queries to the output distribution, as demonstrated by experiments.

Keywords: Probabilistic programs · Quantitative verification · Program equivalence · Denotational semantics · Generating functions

1 Introduction

Probabilistic programs [26,43,48] augment deterministic programs with stochastic behaviors, e.g., random sampling, probabilistic choice, and conditioning (via posterior observations). Probabilistic programs have undergone a recent surge of interest due to prominent applications in a wide range of domains: they steer autonomous robots and self-driving cars [20,54], are key to describe security [6] and quantum [61] mechanisms, intrinsically code up randomized algorithms for solving NP-hard or even deterministically unsolvable problems (in, e.g., distributed computing [2,53]), and are rapidly encroaching on AI as well

This research was funded by the ERC Advanced Project FRAPPANT under grant No. 787914, by the EU's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant No. 101008233, and by the DFG RTG 2236 UnRAVeL.

S. Shoham and Y. Vizel (Eds.): CAV 2022, LNCS 13371, pp. 79–101, 2022.
https://doi.org/10.1007/978-3-031-13185-1_5

as approximate computing [13]. See [5] for recent advancements in probabilistic programming.

The crux of probabilistic programming, à la Hicks' interpretation [30], is to *treat normal-looking programs as if they were probability distributions*. A random-number generator, for instance, is a probabilistic program that produces a uniform distribution across numbers from a range of interest. Such a lift from deterministic program states to possibly infinite-support distributions (over states) renders the verification problem of probabilistic programs notoriously hard [39]. In particular, reasoning about probabilistic loops often amounts to computing quantitative fixed-points which are highly intractable in practice. As a consequence, existing techniques are mostly concerned with approximations, i.e., they strive for verifying or obtaining upper and/or lower bounds on various quantities like assertion-violation probabilities [59], preexpectations [9,28], moments [58], expected runtimes [40], and concentrations [15,16], which reveal only partial information about the probability distribution carried by the program.

In this paper, we address the problem of *how to determine whether a (possibly infinite-state) probabilistic program yields exactly the desired (possibly infinite-support) distribution under all possible inputs*. We highlight two scenarios where encoding the *exact* distribution – other than (bounds on) the above-mentioned quantities – is of particular interest: (I) In many safety- and/or security-critical domains, e.g., cryptography, a slightly perturbed distribution (while many of its probabilistic quantities remain unchanged) may lead to significant attack vulnerabilities or even complete compromise of the cryptographic system, see, e.g., Bleichenbacher's biased-nonces attack [29, Sect. 5.10] against the probabilistic Digital Signature Algorithm. Therefore, the system designer has to impose a complete specification of the anticipated distribution produced by the probabilistic component. (II) In the context of quantitative verification, the user may be interested in multiple properties (of different types, e.g., the aforementioned quantities) of the output distribution carried by a probabilistic program. In absence of the exact distribution, multiple analysis techniques – tailored to different types of properties – have to be applied in order to answer all queries from the user. We further motivate our problem using a concrete example as follows.

Example 1 (Photorealistic Rendering [37]*).* Monte Carlo integration algorithms form a well-known class of probabilistic programs which approximate complex integral expressions by sampling [27]. One of its particular use-cases is the photorealistic rendering of virtual scenes by a technique called *Monte Carlo path tracing* (MCPT) [37].

MCPT works as follows: For every pixel of the output image, it shoots n sample rays into the scene and models the light transport behavior to approximate the incoming light at that particular point. Starting from a certain pixel position, MCPT randomly chooses a direction, traces it until a scene object is hit, and then proceeds by either (i) terminating the tracing and evaluating the overall ray, or (ii) continuing the tracing by computing a new direction. In the physical world, the light ray may be reflected arbitrarily often and thus stopping the tracing after a certain amount of bounces would introduce a bias in the

```
while (n > 0) { /* generate n samples */
    running := 1;
    while (running = 1) { /* generate a light ray */
        { running := 0 /* absorb */ } [1/2] { c := c + 1 /* reflect */ } };
    n := n − 1 }
```

Fig. 1. Monte Carlo path tracing in a scene with constant reflectivity 1/2.

integral estimation. As a remedy, the decision when to stop the tracing is made in a *Russian roulette* manner by flipping a coin[1] at each intersection point [1].

The program in Fig. 1 is an implementation of a simplified MCPT path generator. The cumulative length of all `n` rays is stored in the (random) variable `c`, which is directly proportional to MCPT's expected runtime. The implementation is designed in a way that `c` *induces a distribution as the sum of* `n` *independent and identically distributed (i.i.d.) geometric random variables* such that the resulting integral estimation is unbiased. In our framework, we view such an exact output distribution of `c` as a *specification* and verify – fully automatically – that the implementation in Fig. 1 with nested loops indeed satisfies this specification. ◁

Approach. Given a probabilistic loop $L = \texttt{while}\,(\varphi)\,\{P\}$ with guard φ and loop-free body P, we aim to determine whether L agrees with a specification S:

$$L = \texttt{while}\,(\varphi)\,\{P\} \quad \overset{?}{\sim} \quad S\,, \tag{$\star$}$$

namely, whether L yields – upon termination – exactly the same distribution as encoded by S under all possible program inputs. This problem is non-trivial: (C1) L may induce an infinite state space and infinite-support distributions, thus making techniques like probabilistic bounded model checking [34] insufficient for verifying the property by means of unfolding the loop L. (C2) There is, to the best of our knowledge, a lack of non-trivial characterizations of L and S such that problem ($\star$) admits a decidability result. (C3) To decide problem ($\star$) – even for a loop-free program L – one has to account for infinitely or even uncountably many inputs such that L yields the same distribution as encoded by S when being deployed in all possible contexts.

We address challenge (C1) by exploiting the forward denotational semantics of probabilistic programs based on *probability generating function* (PGF) representations of (sub-)distributions [42], which benefits crucially from closed-form (i.e., finite) PGF representations of possibly infinite-support distributions. A probabilistic program L hence acts as a transformer $[\![L]\!](\cdot)$ that transforms an input PGF g into an output PGF $[\![L]\!](g)$ (as an instantiation of Kozen's

[1] The bias of the coin depends on the material's *reflectivity*: a reflecting material such as a mirror requires more light bounces than an absorptive one, e.g., a black surface.

transformer semantics [43]). In particular, we *interpret the specification S as a loop-free probabilistic program I*. Such an identification of specifications with programs has two important advantages: (i) we only need a single language to encode programs as well as specifications, and (ii) it enables compositional reasoning in a straightforward manner, in particular, the treatment of nested loops. The problem of checking $L \sim S$ then boils down to checking whether L and I transform every possible input PGF into the same output PGF:

$$\forall g \in \mathsf{PGF}\colon \quad \underbrace{[\![\texttt{while}\,(\varphi)\,\{P\}]\!]}_{L}(g) \quad \overset{?}{=} \quad [\![I]\!](g)\ . \tag{†}$$

As I is loop free, problem (†) can be reduced to checking the equivalence of two *loop-free* probabilistic programs (cf. Lemma 2):

$$\forall g \in \mathsf{PGF}\colon \quad [\![\texttt{if}\,(\varphi)\,\{P \fatsemi I\}\,\texttt{else}\,\{\texttt{skip}\}]\!](g) \quad \overset{?}{=} \quad [\![I]\!](g)\ . \tag{‡}$$

Now challenge (C3) applies since the universal quantification in problem (‡) requires to determine the equivalence against infinitely many – possibly infinite-support – distributions over program states. We facilitate such an equivalence checking by developing a *second-order PGF* (SOP) semantics for probabilistic programs, which naturally extends the PGF semantics while allowing to reason about infinitely many PGF transformations simultaneously (see Lemma 3).

Finally, to obtain a decidability result (cf. challenge (C2)), we develop the *rectangular discrete probabilistic programming language* (ReDiP) – a variant of pGCL [46] with syntactic restrictions to rectangular guards – featuring various nice properties, e.g., they inherently support i.i.d. sampling, and in particular, they *preserve closed-form PGF* when acting as PGF transformers. We show that *problem* (‡) *is decidable for* ReDiP *programs P and I if all the distribution statements therein have rational closed-form PGF* (cf. Lemma 4). As a consequence, problem (†) *and thereby* problem (⋆) *of checking $L \sim S$ are decidable if L terminates almost-surely on all possible inputs g* (cf. Theorem 4).

Demonstration. We have automated our techniques in a tool called Prodigy. As an example, Prodigy was able to verify, fully automatically in 25 milliseconds, that the implementation of the MCPT path generator with nested loops (in Fig. 1) is indeed equivalent to the loop-free program

$$\texttt{c} \mathrel{+}= \texttt{iid}(\texttt{geometric}(1/2), \texttt{n}) \fatsemi \texttt{n} := 0$$

which encodes the specification that, upon termination, c is distributed as the sum of n i.i.d. geometric random variables. With such an output distribution, multiple queries can be efficiently answered by applying standard PGF operations. For example, the expected value and variance of the runtime are $E[\texttt{c}] = n$ and $Var[\texttt{c}] = 2n$, respectively (assuming $\texttt{c} = 0$ initially).

Contributions. The main contributions of this paper are:

- The probabilistic programming language `ReDiP` and its forward denotational semantics as PGF transformers. We show that loop-free `ReDiP` programs preserve closed-form PGF.
- The notion of SOP that enables reasoning about infinitely many PGF transformations simultaneously. We show that the problem of determining whether an infinite-state `ReDiP` loop generates – upon termination – exactly a specified distribution is decidable.
- The software tool Prodigy which supports automatic invariance checking on the source-code level; it allows reasoning about nested `ReDiP` loops in a compositional manner, and supports efficient queries on various quantities including assertion-violation probabilities, expected values, (high-order) moments, precise tail probabilities, as well as concentration bounds.

Organization. We introduce generating functions in Sect. 2 and define the `ReDiP` language in Sect. 3. Section 4 presents the PGF semantics. Section 5 establishes our decidability result in reasoning about `ReDiP` loops, with case studies in Sect. 6. After discussing related work in Sect. 7, we conclude the paper in Sect. 8. Further details, e.g., proofs and additional examples, can be found in the full version [18].

2 Generating Functions

> *"A generating function is a clothesline on which we hang up a sequence of numbers for display."* — H. S. Wilf, Generatingfunctionology [60]

The method of *generating functions* (GF) is a vital tool in many areas of mathematics. This includes in particular enumerative combinatorics [22,60] and – most relevant for this paper – probability theory [35]. In the latter, the sequences "hanging on the clotheslines" happen to describe probability distributions over the non-negative integers $\mathbb{N}$, e.g., $1/2, 1/4, 1/8, \ldots$ (aka, the geometric distribution).

The most common way to relate an (infinite) *sequence* of numbers to a generating *function* relies on the familiar Taylor series expansion: Given a sequence, for example $1/2, 1/4, 1/8, \ldots$, find a function $x \mapsto f(x)$ whose Taylor series around $x = 0$ uses the numbers in the sequence as coefficients. In our example,

$$\frac{1}{2-x} \;=\; \frac{1}{2} + \frac{1}{4}x + \frac{1}{8}x^2 + \frac{1}{16}x^3 + \frac{1}{32}x^4 + \ldots, \tag{1}$$

for all $|x| < 2$, hence the "clothesline" used for hanging up $1/2, 1/4, 1/8, \ldots$ is the function $1/(2-x)$. Note that the GF is a – from a purely syntactical point of view – *finite* object while the sequence it represents is *infinite*. A key strength of this technique is that many meaningful operations on infinite series can be performed by manipulating an encoding GF (see Table 1 for an overview and examples). In other words, GF provide an *interface* to perform operations on and extract information from infinite sequences in an effective manner.

2.1 The Ring of Formal Power Series

Towards our goal of encoding distributions over *program states* (valuations of finitely many integer variables) as generating functions, we need to consider *multivariate* GF, i.e., GF with more than one variable. Such functions represent multidimensional sequences, or *arrays*. Since multidimensional Taylor series quickly become unhandy, we will follow a more *algebraic* approach that is also advocated in [60]: We treat sequences and arrays as elements from an algebraic structure: the *ring of Formal Power Series* (FPS). Recall that a (commutative) *ring* $(A, +, \cdot, 0, 1)$ consists of a non-empty carrier set A, associative and commutative binary operations "+" (addition) and "$\cdot$" (multiplication) such that multiplication distributes over addition, and neutral elements 0 and 1 w.r.t. addition and multiplication, respectively. Further, every $a \in A$ has an additive inverse $-a \in A$. Multiplicative inverses $a^{-1} = 1/a$ need not always exist. Let $k \in \mathbb{N} = \{0, 1, \ldots\}$ be fixed in the remainder.

Table 1. GF cheat sheet. f, g and X, Y are arbitrary GF and indeterminates, resp.

Operation	*Effect*	*(Running) example*
$f^{-1} = 1/f$	Multiplicative inverse of f (if it exists)	$\frac{1}{1-XY} = 1 + XY + X^2Y^2 + \ldots$ because $(1 - XY)(1 + XY + X^2Y^2 + \ldots) = 1$
fX	Shift in dimension X	$\frac{X}{1-XY} = X + X^2Y + X^3Y^2 + \ldots$
$f[X/0]$	Drop terms containing X	$\frac{1}{1-0Y} = 1$
$f[X/1]$	Projection[a] on Y	$\frac{1}{1-1Y} = 1 + Y + Y^2 + \ldots$
fg	Discrete convolution (or Cauchy product)	$\frac{1}{(1-XY)^2} = 1 + 2XY + 3X^2Y^2 + \ldots$
$\partial_X f$	Formal derivative in X	$\partial_X \frac{1}{1-XY} = \frac{Y}{(1-XY)^2} = Y + 2XY^2 + 3X^2Y^3 + \ldots$
$f + g$	Coefficient-wise sum	$\frac{1}{1-XY} + \frac{1}{(1-XY)^2} = \frac{2-XY}{(1-XY)^2} = 2{+}3XY{+}4X^2Y^2{+}\ldots$
af	Coefficient-wise scaling	$\frac{7}{(1-XY)^2} = 7 + 14XY + 21X^2Y^2 + \ldots$

[a] Projections are not always well-defined, e.g., $\frac{1}{1-X+Y}[X/1] = \frac{1}{Y}$ is ill-defined because Y is not invertible. However, in all situations where we use projection it will be well-defined; in particular, projection is well-defined for PGF.

Definition 1 (The Ring of FPS). *A k-dimensional FPS is a k-dim. array $f\colon \mathbb{N}^k \to \mathbb{R}$. We denote FPS as* formal sums *as follows: Let $\mathbf{X}=(X_1, \ldots, X_k)$ be an ordered vector of symbols, called* indeterminates*. The FPS f is written as*

$$f \;=\; \sum\nolimits_{\sigma \in \mathbb{N}^k} f(\sigma)\mathbf{X}^\sigma$$

where $\mathbf{X}^\sigma$ is the monomial $X_1^{\sigma_1} X_2^{\sigma_2} \cdots X_k^{\sigma_k}$*. The* ring of FPS *is denoted $\mathbb{R}[[\mathbf{X}]]$ where the operations are defined as follows: For all $f, g \in \mathbb{R}[[\mathbf{X}]]$ and $\sigma \in \mathbb{N}^k$, $(f + g)(\sigma) = f(\sigma) + g(\sigma)$, and $(f \cdot g)(\sigma) = \sum_{\sigma_1+\sigma_2=\sigma} f(\sigma_1)g(\sigma_2)$.*

The multiplication $f \cdot g$ is the usual *Cauchy product* of power series (aka discrete convolution); it is well defined because for all $\sigma \in \mathbb{N}^k$ there are just *finitely* many $\sigma_1 + \sigma_2 = \sigma$ in $\mathbb{N}^k$. We write fg instead of $f \cdot g$.

The formal sum notation is standard in the literature and often useful because the arithmetic FPS operations are very similar to how one would do calculations with "real" sums. We stress that the indeterminates $\mathbf{X}$ are merely *labels* for the k dimensions of f and do not have any other particular meaning. In the context of this paper, however, it is natural to identify the indeterminates with the program variables (e.g. indeterminate X refers to variable $\mathtt{x}$, see Sect. 3).

Equation (1) can be interpreted as follows in the ring of FPS: The "sequences" $2 - 1X + 0X^2 + \dots$ and $1/2 + 1/4X + 1/8X^2 + \dots$ are (multiplicative) *inverse* elements to each other in $\mathbb{R}[[X]]$, i.e., their product is 1. More generally, we say that an FPS f is *rational* if $f = gh^{-1} = g/h$ where g and h are polynomials, i.e., they have at most finitely many non-zero coefficients; and we call such a representation a *rational closed form*.

A more extensive introduction to FPS can be found in [18, Appx. D].

2.2 Probability Generating Functions

We are especially interested in GF that describe probability distributions.

Definition 2 (PGF). *A k-dimensional FPS g is a* probability generating function *(PGF) if* (i) *for all $\sigma \in \mathbb{N}^k$ we have $g(\sigma) \geq 0$, and* (ii) $\sum_{\sigma \in \mathbb{N}^k} g(\sigma) \leq 1$.

For example, (1) is the PGF of a $1/2$-geometric distribution. The PGF of other standard distributions are given in Table 3 further below. Note that Definition 2 also includes *sub-PGF* where the sum in (ii) is strictly less than 1.

3 ReDiP: A Probabilistic Programming Language

This section presents our *Rectangular Discrete Probabilistic Programming Language*, or ReDiP for short. The word "rectangular" refers to a restriction we impose on the guards of conditionals and loops, see Sect. 3.2. ReDiP is a variant of pGCL [46] with some extra syntax but also some syntactic restrictions.

3.1 Program States and Variables

Every ReDiP-program P operates on a finite set of $\mathbb{N}$-valued *program variables* $Vars(P) = \{\mathtt{x}_1, \dots, \mathtt{x}_k\}$. We do not consider negative or non-integer variables. A *program state* of P is thus a mapping $\sigma \colon Vars(P) \to \mathbb{N}$. As explained in Sect. 1, the key idea is to represent distributions over such program states as PGF. Consequently, we identify a single program state σ with the *monomial* $\mathbf{X}^\sigma = X_1^{\sigma(\mathtt{x}_1)} \cdots X_k^{\sigma(\mathtt{x}_k)}$ where $X_1, \dots, X_k$ are indeterminates representing the program variables $\mathtt{x}_1, \dots, \mathtt{x}_k$. We will stick to this notation: throughout the whole paper, we typeset program variables as $\mathtt{x}$ and the corresponding FPS indeterminate as X. The initial program state on which a given ReDiP-program is supposed to operate must always be stated explicitly.

3.2 Syntax of ReDiP

The syntax of ReDiP is defined inductively, see the leftmost column of Table 2. Here, x and y are program variables, $n \in \mathbb{N}$ is a constant, D is a *distribution expression* (see Table 3), and P_1, P_2 are ReDiP-programs. The general idea of ReDiP is to provide a minimal core language to keep the theory simple. Many other common language constructs such as linear arithmetic updates $\mathtt{x} := 2\mathtt{y} + 3$ are expressible in this core language. See [18, Appx. A] for a complete specification.

Table 2. Syntax and semantics of ReDiP. g is the input PGF.

ReDiP-*program* P	*Semantics* $[\![P]\!](g)$ – *see Sect.* 4.2	*Description*
$\mathtt{x} := n$	$g[X/1]X^n$	Assign const. $n \in \mathbb{N}$ to var. x
$\mathtt{x}{-}{-}$	$(g - g[X/0])X^{-1} + g[X/0]$	Decr. x ("monus" semantics)
$\mathtt{x} \mathrel{+}= \mathtt{iid}(D, \mathtt{y})$	$g[Y/Y[\![D]\!][T/X]]$	Incr. x by the sum of y i.i.d. samples from D – see Sect. 3.3
$\mathtt{if}\ (\mathtt{x} < n)\ \{P_1\}$ $\mathtt{else}\ \{P_2\}$	$[\![P_1]\!](g_{x<n}) + [\![P_2]\!](g - g_{x<n})$, where $g_{x<n} = \sum_{i=0}^{n-1} \frac{1}{i!}(\partial_X^i g)[X/0]X^i$	Conditional branching
$P_1 ; P_2$	$[\![P_2]\!]([\![P_1]\!](g))$	Sequential composition
$\mathtt{while}\ (\mathtt{x} < n)\ \{P_1\}$	$\left[\mathsf{lfp}\ \Psi_{\mathtt{x}<n,P_1}\right](g)$, where $\Psi_{\mathtt{x}<n,P_1}(\psi) =$ $\lambda f.\ (f - f_{\mathtt{x}<n}) + \psi([\![P_1]\!](f_{\mathtt{x}<n}))$	Loop defined as fixed point

Table 3. A non-exhaustive list of common discrete distributions with rational PGF. The parameters p, n, and λ are a probability, a natural, and a non-negative real number, respectively. T is a reserved placeholder indeterminate.

D	$[\![D]\!]$	*Description*
$\mathtt{dirac}(n)$	T^n	Point mass
$\mathtt{bernoulli}(p)$	$1 - p + pT$	Bernoulli distribution (coin flip)
$\mathtt{unif}(n)$	$(1 - T^n)\,/\,n(1 - T)$	Discrete uniform distribution on $\{0, \ldots, n{-}1\}$
$\mathtt{geometric}(p)$	$(1 - p)\,/\,(1 - pT)$	Geometric distribution (no. trials until first success)
$\mathtt{binomial}(p,\ n)$	$(1 - p + pT)^n$	Binomial distribution (successes of n yes-no experiments)
$\mathtt{nbinomial}(p,\ n)$	$(1 - p)^n\,/\,(1 - pT)^n$	Negative binomial distribution

The word "rectangular" in ReDiP emphasizes that our if-guards can only identify *axis-aligned hyper-rectangles*[2] in $\mathbb{N}^k$, but no more general polyhedra. These *rectangular guards* $\mathtt{x} < n$ have the fundamental property that they preserve rational PGF. On the other hand, allowing more general guards like $\mathtt{x} < \mathtt{y}$ breaks this property (see [21] and our comments in [18, Appx. B].

The most intricate feature of ReDiP is the – potentially unbounded – loop $\mathtt{while}\ (\mathtt{x} < n)\ \{P\}$. A program that does not contain loops is called *loop-free*.

[2] More precisely, we can simulate statements like $\mathtt{if}\ (R)\ \{...\}\ \mathtt{else}\ \{...\}$, where R is a finite Boolean combination of rectangular guards, through appropriate nesting of $\mathtt{if}\ ()$; note that such an R is indeed a finite union of axis-aligned rectangles in $\mathbb{N}^k$.

3.3 The Statement x += iid(D, y)

The novel iid statement is the heart of the loop-free fragment of ReDiP – it subsumes both x := D ("assign a D-distributed sample to x") and the standard assignment x := y. We include the assign-increment (+=) version of iid in the core fragment of ReDiP for technical reasons; the assignment x := iid(D, y) can be recovered from that as syntactic sugar by simply setting x := 0 beforehand.

Intuitively, the meaning of x += iid(D, y) is as follows. The right-hand side iid(D, y) can be seen as a function that takes the current value v of variable y, then draws v i.i.d. samples from distribution D, computes the sum of all these samples and finally increments x by the so-obtained value. For example, to perform x := y, we may just write x := iid(dirac(1), y) as this will draw y times the number 1, then sum up these y many 1's to obtain the result y and assign it to x. Similarly, to assign a random sample from a, say, uniform distribution to x, we can execute y := 1 ; x := iid(unif(n), y).

But iid is not only useful for defining standard operations. In fact, taking sums of i.i.d. samples is common in probability theory. The *binomial distribution* with parameters $p \in (0,1)$ and $n \in \mathbb{N}$, for example, is the defined as the sum of n i.i.d. Bernoulli-p-distributed samples and thus

x := binomial(p, y) is equivalent to x := iid(bernoulli(p), y)

for all constants $p \in (0,1)$. Similarly, the *negative* (p,n)-binomial distribution is the sum of n i.i.d. geometric-p-distributed samples. Overall, iid renders the loop-free fragment of ReDiP *strictly more expressive* than it would be if we had included only x := D and x := y instead. As a consequence, since we use loop-free programs as a specification language (see Sect. 5), iid enables us to write more expressive program specifications while retaining decidability.

4 Interpreting ReDiP with PGF

In this section, we explain the PGF-based semantics of our language which is given in the second column of Table 2. The overall idea is to view a ReDiP-program P as a *distribution transformer* [44,46]. This means that the input to P is a *distribution* over initial program states (inputting a deterministic state is just the special case of a Dirac distribution), and the output is a distribution over final program states. With this interpretation, if one regards distributions as *generalized program states* [33], a probabilistic program is actually *deterministic*: The same input distribution always yields the same output distribution. The goal of our PGF-based semantics is to construct an *interpreter* that executes a ReDiP-program statement-by-statement in forward direction, transforming one generalized program state into the next. We stress that these generalized program states, or distributions, can be infinite-support in general. For example, the program x := geometric(0.5) outputs a geometric distribution – which has infinite support – on x.

4.1 A Domain for Distribution Transformation

We now define a domain, i.e., an *ordered* structure, where our program's in- and output distributions live. Following the general idea of this paper, we encode them as PGF. Let *Vars* be a fixed finite set of program variables $\mathtt{x}_1, \ldots, \mathtt{x}_k$ and let $\mathbf{X} = (X_1, \ldots, X_k)$ be corresponding formal indeterminates. We let $\mathsf{PGF} = \{g \in \mathbb{R}[[\mathbf{X}]] \mid g \text{ is a PGF}\}$ denote the set of all PGF. Recall that this also includes sub-PGF (Definition 2). Further, we equip PGF with the pointwise order, i.e., we let $g \sqsubseteq f$ iff $g(\sigma) \leq f(\sigma)$ for all $\sigma \in \mathbb{N}^k$. It is clear that $(\mathsf{PGF}, \sqsubseteq)$ is a partial order that is moreover ω-*complete*, i.e., there exists a least element 0 and all ascending chains $\Gamma = \{g_0 \sqsubseteq g_1 \sqsubseteq \ldots\}$ in PGF have a least upper bound $\sup \Gamma \in \mathsf{PGF}$. The maxima in $(\mathsf{PGF}, \sqsubseteq)$ are precisely the PGF which are not a sub-PGF.

4.2 From Programs to PGF Transformers

Next we explain how distribution transformation works using (P)GF (cf. Table 1). This is in contrast to the PGF semantics from [42] which operates on infinite sums in a non-constructive fashion.

Definition 3 (The PGF Transformer $[\![P]\!]$). *Let P be a* `ReDiP`*-program. The* PGF transformer $[\![P]\!] \colon \mathsf{PGF} \to \mathsf{PGF}$ *is defined inductively on the structure of P through the second column in Table 2.*

We show in Theorem 2 below that $[\![P]\!]$ is well-defined. For now, we go over the statements in the language `ReDiP` and explain the semantics.

Sequential Composition. The semantics of $P_1 ; P_2$ is straightforward and intuitive: First execute P_1 on g and then P_2 on $[\![P_1]\!](g)$, i.e., $[\![P_1 ; P_2]\!](g) = [\![P_2]\!]([\![P_1]\!](g))$. The fact that our semantics transformer moves *forwards* through the program – as program interpreters usually do – is due to this definition.

Conditional Branching. To translate $\mathtt{if}\ (\mathtt{x} < n)\ \{P_1\}\ \mathtt{else}\ \{P_2\}$, we follow the standard procedure which partitions the input distribution according to $\mathtt{x} < n$ and $\mathtt{x} \geq n$, processes the two parts independently and finally recombines the results [44]. We realize the partitioning using the (formal) *Taylor series expansion*. This is feasible because we only allow *rectangular* guards of the form $\mathtt{x} < n$, where n is a constant. Thus, for a given input PGF g, the *filtered PGF* $g_{\mathtt{x}<n}$ is obtained through expanding g in its first n terms. The `else` -part is obviously $g_{\mathtt{x}\geq n} = g - g_{\mathtt{x}<n}$. We then evaluate $[\![P_1]\!](g_{\mathtt{x}<n}) + [\![P_2]\!](g_{\mathtt{x}\geq n})$ recursively.

Assigning a Constant. Technically, our semantics realizes an assignment $\texttt{x} := n$ in two steps: It first sets $\texttt{x}$ to 0 and then increments it by n. The former is achieved by substituting X for 1 which corresponds to computing the marginal distribution in all variables except X. For example,

$$
\begin{array}{ll}
/\!/\!/\ 0.5XY^2 + 0.5X^2Y^3 & /\!/\!/\ g \\
\texttt{x} := 5 & P \\
/\!/\!/\ (0.5Y^2 + 0.5Y^3)X^5 & /\!/\!/\ [\![P]\!](g) \\
/\!/\!/\ 0.5X^5Y^2 + 0.5X^5Y^3 & /\!/\!/\ \langle \textit{reform. of prev. line} \rangle
\end{array}
$$

where the rightmost four lines explain this annotation style [42]. Note that $0.5Y^2 + 0.5Y^3$ is indeed the marginal of the input distribution in Y.

Decrementing a Variable. Since our program variables cannot take negative values, we define $\texttt{x}{-}{-}$ as $\max(\texttt{x}-1, 0)$, i.e., $\texttt{x}$ *monus* (modified minus) 1. Technically, we realize this through $\texttt{if}\ (\texttt{x} < 1)\ \{\texttt{skip}\}\ \texttt{else}\ \{\texttt{x}{-}{-}\}$, i.e., we apply the decrement only to the portion of the input distribution where $\texttt{x} \geq 1$. The decrement itself can then be carried out through "multiplication by X^{-1}". Note that X^{-1} is not an element of $\mathbb{R}[[X]]$ because X has no inverse. Instead, the operation gX^{-1} is an alias for $\mathit{shift}^{\leftarrow}(g)$ which shifts g "to the left" in dimension X. To implement the semantics on top of existing computer algebra software, it is very handy to perform the multiplication by X^{-1} instead. This is justified because for PGF g with $g[X/0] = 0$, $\mathit{shift}^{\leftarrow}(g)$ and gX^{-1} are equal.

The `iid` *Statement.* The semantics of $\texttt{x} \mathrel{+}= \texttt{iid}(D, \texttt{y})$ relies on the fact that

$$
T_1 \sim [\![D]\!]\ \ldots\ T_n \sim [\![D]\!] \qquad \text{implies} \qquad \sum\nolimits_{i=1}^{n} T_i \sim [\![D]\!]^n\ , \tag{2}
$$

where $X \sim g$ means that r.v. X is distributed according to PGF g (see, e.g., [55, p. 450]). The `iid` statement generalizes this observation further: If n is not a constant but a random (program) variable $\texttt{y}$ with PGF $h(Y)$, then we perform the *substitution* $h[Y/[\![D]\!]]$ (i.e., replace Y by $[\![D]\!]$ in h) to obtain the PGF of the sum of $\texttt{y}$-many i.i.d. samples from D. We slightly modify this substitution to $g[Y/Y[\![D]\!][T/X]]$ in order to (i) not alter $\texttt{y}$, and (ii) account for the increment to $\texttt{x}$. For example,

$$
\begin{array}{l}
/\!/\!/\ 0.2 + 0.3Y + 0.5Y^2 \\
\texttt{x} \mathrel{+}= \texttt{iid}(\texttt{bernoulli}(0.5), \texttt{y}) \\
/\!/\!/\ 0.2 + 0.3Y(0.5 + 0.5X) + 0.5Y^2(0.5 + 0.5X)^2 \\
/\!/\!/\ 0.2 + 0.15Y + 0.125Y^2 + 0.15XY + 0.25XY^2 + 0.125X^2Y^2\ .
\end{array}
$$

The `while`*-Loop.* The fixed point semantics of the while loop is standard [42,44] and reflects the intuitive *unrolling rule*, namely that $\texttt{while}\ (\varphi)\ \{P\}$ is equivalent to $\texttt{if}\ (\varphi)\ \{P\fatsemi\ \texttt{while}\ (\varphi)\ \{P\}\}\ \texttt{else}\ \{\texttt{skip}\}$. Indeed, the fixed point formula in

Table 2 can be derived using the semantics of `if` discussed above. We revisit this fixed point characterization in Sect. 5.1.

Properties of $[\![P]\!]$. Our PGF semantics has the property that all programs – except while loops – are able to operate on the input PGF in (rational) *closed form*, i.e., they never have to expand the input as an infinite series (which is of course impossible in practice). More formally:

Theorem 1 (Closed-Form Preservation). *Let P be a* loop-free `ReDiP` *program, and let $g = h/f \in \mathsf{PGF}$ be in rational closed form. Then we can compute a rational closed form of $[\![P]\!](g) \in \mathsf{PGF}$ by applying the transformations in Table 2.*

The proof is by induction over the structure of P noticing that all the necessary operations (substitution, differentiation, etc.) preserve rational closed forms, see [18, Appx. D]. A slight extension of our syntax, e.g., admitting non-rectangular guards, renders that closed forms are not preserved, see [18, Appx. B]. Moreover, $[\![P]\!]$ has the following *healthiness* [46] properties:

Theorem 2 (Properties of $[\![P]\!]$). *The PGF transformer $[\![P]\!]$ is*

- *a well-defined function* $\mathsf{PGF} \to \mathsf{PGF}$,
- continuous, *i.e.,* $[\![P]\!](\sup \Gamma) = \sup [\![P]\!](\Gamma)$ *for all chains* $\Gamma \subseteq \mathsf{PGF}$,
- linear, *i.e.,* $[\![P]\!](\sum_{\sigma \in \mathbb{N}^k} g(\sigma)\mathbf{X}^\sigma) = \sum_{\sigma \in \mathbb{N}^k} g(\sigma)[\![P]\!](\mathbf{X}^\sigma)$ *for all* $g \in \mathsf{PGF}$.

4.3 Probabilistic Termination

Due to the presence of possibly unbounded `while`-loops, a `ReDiP`-program does not necessarily halt, or may do so only with a certain probability. Our semantics naturally captures the termination probability.

Definition 4 (AST). *A* `ReDiP`*-program P is called* almost-surely terminating *(AST) for PGF g if $[\![P]\!](g)[\mathbf{X}/\mathbf{1}] = g[\mathbf{X}/\mathbf{1}]$, i.e., if it does not leak probability mass. P is called* universally *AST (UAST) if it is AST for all $g \in \mathsf{PGF}$.*

Note that all loop-free `ReDiP`-programs are UAST. In this paper, (U)AST only plays a minor role. Nonetheless, the proof rule below yields a stronger result (cf. Lemma 2) if the program is UAST. There exist various of techniques and tools for proving (U)AST [17,47,50].

5 Reasoning About Loops

We now focus on loopy programs $L = \texttt{while}\,(\varphi)\,\{P\}$. Recall from Table 2 that $[\![L]\!] \colon \mathsf{PGF} \to \mathsf{PGF}$ is defined as the *least fixed point* of a higher order functional

$$\Psi_{\varphi,P} \colon (\mathsf{PGF} \to \mathsf{PGF}) \;\to\; (\mathsf{PGF} \to \mathsf{PGF}).$$

Following [42], we show that $\Psi_{\varphi,P}$ is sufficiently well-behaved to allow reasoning about loops by *fixed point induction*.

5.1 Fixed Point Induction

To apply fixed point induction, we need to lift our domain PGF from Sect. 4.1 by one order to $(\mathsf{PGF} \to \mathsf{PGF})$, the domain of *PGF transformers*. This is because the functional $\Psi_{\varphi,P}$ operates on PGF transformers and can thus be seen as a second-order function (this point of view regards PGF as first-order objects). Recall that in contrast to this, the function $[\![P]\!]$ is first-order – it is just a PGF transformer. The order on $(\mathsf{PGF} \to \mathsf{PGF})$ is obtained by lifting the order $\sqsubseteq$ on PGF pointwise (we denote it with the same symbol $\sqsubseteq$). This implies that $(\mathsf{PGF} \to \mathsf{PGF})$ is also an ω-complete partial order. We can then show that $\Psi_{\varphi,P}$ (see Table 2) is a continuous function. With these properties, we obtain the following induction rule for upper bounds on $[\![L]\!]$, cf. [42, Theorem 6]:

Lemma 1 (Fixed Point Induction for Loops). *Let* $L = \texttt{while}\,(\varphi)\,\{P\}$ *be a* ReDiP*-loop. Further, let* $\psi\colon \mathsf{PGF} \to \mathsf{PGF}$ *be a PGF transformer. Then*

$$\Psi_{\varphi,P}(\psi) \;\sqsubseteq\; \psi \qquad \text{implies} \qquad [\![L]\!] \;\sqsubseteq\; \psi\ .$$

The goal of the rest of the paper is to *apply the rule from* Lemma 1 in *practice*. To this end, we must somehow specify an *invariant* such as ψ by finite means. Since ψ is of type $(\mathsf{PGF} \to \mathsf{PGF})$, we consider ψ as a program I – more specifically, a ReDiP-program – and identify $\psi = [\![I]\!]$. Further, by definition

$$\Psi_{\varphi,P}([\![I]\!]) \;=\; [\![\texttt{if}\,(\varphi)\,\{P\,\texttt{;}\,I\}\,\texttt{else}\,\{\texttt{skip}\}]\!],$$

and thus the term $\Psi_{\varphi,P}([\![I]\!])$ is also a PGF-transformer expressible as a ReDiP-program. These observations and Lemma 1 imply the following:

Lemma 2. *Let* $L = \texttt{while}\,(\varphi)\,\{P\}$ *and* I *be* ReDiP*-programs. Then*

$$[\![\texttt{if}\,(\varphi)\,\{P\,\texttt{;}\,I\}\,\texttt{else}\,\{\texttt{skip}\}]\!] \;\sqsubseteq\; [\![I]\!] \quad \text{implies} \quad [\![L]\!] \;\sqsubseteq\; [\![I]\!]. \tag{3}$$

Further, if L *is UAST (Definition 4), then*

$$[\![\texttt{if}\,(\varphi)\,\{P\,\texttt{;}\,I\}\,\texttt{else}\,\{\texttt{skip}\}]\!] \;=\; [\![I]\!] \quad \text{iff} \quad [\![L]\!] \;=\; [\![I]\!] \tag{4}$$

Lemma 2 effectively reduces checking whether ψ given as a ReDiP-program I is an invariant of L to checking *equivalence* of $\texttt{if}\,(\varphi)\,\{P\,\texttt{;}\,I\}\,\texttt{else}\,\{\texttt{skip}\}$ and I provided L is UAST. If I is loop-free, then the latter two programs are both loop-free and we are left with the task of proving whether they yield the same output distribution for all inputs. We now present a solution to this problem.

5.2 Deciding Equivalence of Loop-free Programs

Even in the absence of loops, deciding if two given ReDiP-programs are equivalent is non-trivial as it requires reasoning about infinitely many – possibly infinite-support – distributions on program variables. In this section, we first show that $[\![P_1]\!] = [\![P_2]\!]$ is *decidable* for loop-free ReDiP programs P_1 and P_2, and then use this result together with Lemma 2 to obtain the main result of this paper.

SOP: Second-Order PGF. Our goal is to check if $[\![P_1]\!](g) = [\![P_2]\!](g)$ for *all* $g \in \mathsf{PGF}$. To tackle this, we encode whole *sets* of PGF into a single object – an FPS we call *second-order PGF* (SOP). To define SOP, we need a slightly more flexible view on FPS. Recall from Definition 1 that a k-dim. FPS is an array $f\colon \mathbb{N}^k \to \mathbb{R}$. Such an f can be viewed equivalently as an l-dim. array with $(k-l)$-dim. arrays as entries. In the formal sum notation, this is reflected by partitioning $\mathbf{X} = (\mathbf{Y}, \mathbf{Z})$ and viewing f as an FPS in $\mathbf{Y}$ *with coefficients that are FPS in the other indeterminates* $\mathbf{Z}$. For example,

$$\begin{aligned}(1-Y)^{-1}(1-Z)^{-1} &= 1 + Y + Z + Y^2 + YZ + Z^2 + \dots \\ &= (1-Z)^{-1} + (1-Z)^{-1}Y + (1-Z)^{-1}Y^2 + \dots\end{aligned}$$

where in the lower line the coefficients $(1-Z)^{-1}$ are considered elements in $\mathbb{R}[[Z]]$.

Definition 5 (SOP). *Let* $\mathbf{U}$ *and* $\mathbf{X}$ *be disjoint sets of indeterminates. A formal power series* $f \in \mathbb{R}[[\mathbf{U}, \mathbf{X}]]$ *is a* second-order PGF (SOP) *if*

$$f = \sum\nolimits_{\tau \in \mathbb{N}^{|\mathbf{U}|}} f(\tau)\mathbf{U}^\tau \quad (\textit{with } f(\tau) \in \mathbb{R}[[\mathbf{X}]]) \qquad \text{implies} \qquad \forall \tau\colon f(\tau) \in \mathsf{PGF}.$$

That is, an SOP is simply an FPS whose coefficients are PGF – instead of generating a sequence of probabilities as PGF do, it generates a *sequence of distributions*. An (important) example SOP is

$$f_{dirac} \;=\; (1-XU)^{-1} \;=\; 1 + XU + X^2U^2 + \dots \;\in \mathbb{R}[[U, X]], \tag{5}$$

i.e., for all $i \geq 0$, $f_{dirac}(i) = X^i = [\![\texttt{dirac}(i)]\!]$. As a second example consider $f_{binom} = f_{dirac}[X/0.5 + 0.5X]$; it is clear that $f_{binom}(i) = (0.5 + 0.5X)^i = [\![\texttt{binomial}(0.5,\, i)]\!]$ for all $i \geq 0$. Note that if $\mathbf{U} = \emptyset$, then SOP and PGF coincide. For fixed $\mathbf{X}$ and $\mathbf{U}$, we denote the set of all second-order PGF with SOP.

SOP Semantics of `ReDiP`. The appeal of SOP is that, syntactically, they are still formal power series, and some can be represented in closed form just like PGF. Moreover, we can readily extend our PGF transformer $[\![P]\!]$ to an SOP transformer $[\![P]\!]\colon \mathsf{SOP} \to \mathsf{SOP}$. A key insight of this paper is that – without any changes to the rules in Table 2 – applying $[\![P]\!]$ to an SOP is the same as applying $[\![P]\!]$ *simultaneously* to all the PGF it subsumes:

Theorem 3. *Let* P *be a* `ReDiP`*-program. The transformer* $[\![P]\!]\colon \mathsf{SOP} \to \mathsf{SOP}$ *is well-defined. Further, if* $f = \sum_{\tau \in \mathbb{N}^{|\mathbf{U}|}} f(\tau)\mathbf{U}^\tau$ *is an SOP, then*

$$[\![P]\!](f) \;=\; \sum\nolimits_{\tau \in \mathbb{N}^{|\mathbf{U}|}} [\![P]\!](f(\tau))\mathbf{U}^\tau \;.$$

An SOP Transformation for Proving Equivalence. We now show how to exploit Theorem 3 for equivalence checking. Let P_1 and P_2 be (loop-free) `ReDiP`-programs; we are interested in proving whether $[\![P_1]\!] = [\![P_2]\!]$. By linearity it holds that $[\![P_1]\!] = [\![P_2]\!]$ iff $[\![P_1]\!](\mathbf{X}^\sigma) = [\![P_2]\!](\mathbf{X}^\sigma)$ for all $\sigma \in \mathbb{N}^k$, i.e., to check equivalence it suffices to consider all (infinitely many) point-mass PGF as inputs.

Lemma 3 (SOP-Characterisation of Equivalence). *Let P_1 and P_2 be* ReDiP*-programs with $Vars(P_i) \subseteq \{\mathtt{x}_1, \ldots, \mathtt{x}_k\}$ for $i \in \{1, 2\}$. Further, consider a vector $\mathbf{U} = (U_1, \ldots, U_k)$ of meta indeterminates, and let $g_{\mathbf{X}}$ be the SOP*

$$g_{\mathbf{X}} \;=\; (1 - X_1U_1)^{-1}(1 - X_2U_2)^{-1} \cdots (1 - X_kU_k)^{-1} \;\in\; \mathbb{R}[[\mathbf{U}, \mathbf{X}]] \;.$$

Then $[\![P_1]\!] = [\![P_2]\!]$ if and only if $[\![P_1]\!](g_{\mathbf{X}}) = [\![P_2]\!](g_{\mathbf{X}})$.

The proof of Lemma 3 (see [18, Appx. F.5]) relies on Theorem 3 and the fact that the *rational* SOP $g_{\mathbf{X}}$ generates all (multivariate) point-mass PGF; in fact it holds that $g_{\mathbf{X}} = \sum_{\sigma \in \mathbb{N}^k} \mathbf{X}^\sigma \mathbf{U}^\sigma$, i.e., $g_{\mathbf{X}}$ generalizes f_{dirac} from (5). It follows:

Lemma 4. *$[\![P_1]\!] = [\![P_2]\!]$ is decidable for loop-free* ReDiP*-programs P_1, P_2.*

Our main theorem follows immediately from Lemmas 2 and 4:

Theorem 4. *Let $L =$* `while` $(\varphi)\,\{P\}$ *be UAST with loop-free body P and I be a loop-free* ReDiP*-program. It is decidable whether $[\![L]\!] = [\![I]\!]$.*

Example 2. In Fig. 2 we prove that the two UAST programs L and I

```
while (n > 0) {                          c += iid(geometric(1/2), n);
    { n := n − 1 } [1/2] { c := c + 1 }  n := 0
```

```
/// (1 − NU)^{-1}(1 − CV)^{-1} = g_{N,C} =: g_0
if (n > 0) {
    /// (1 − CV)^{-1}((1 − NU)^{-1} − 1) = g_0 − g_0[N/0] =: g_1
    { n := n − 1
        /// N^{-1}(1 − CV)^{-1}((1 − NU)^{-1} − 1) = g_1 N^{-1} =: g_2
    } [0.5] { c += 1 };
        /// C(1 − CV)^{-1}((1 − NU)^{-1} − 1) = g_1 C =: g_3
    /// (2N(1 − CV))^{-1} + C(2(1 − CV))^{-1})((1 − NU)^{-1} − 1) = 0.5 g_2 + 0.5 g_3 =: g_4
    c += iid(geometric(1/2), n);
    /// (2 − C)(2N(1 − CV))^{-1} + C(2(1 − CV))^{-1})((2 − C)(2 − C − NU)^{-1} − 1)
        = g_4[N/N(2 − C)^{-1}] =: g_5
    n := 0 }
    /// (1 − CV)^{-1}((2 − C)(2 − C − U)^{-1} − 1) = g_5[N/1] = g_6
/// (1 − CV)^{-1}(2 − C)(2 − C − U)^{-1} = g_6 + g_0[N/0]
```

```
/// (1 − NU)^{-1}(1 − CV)^{-1}
c += iid(geometric(1/2), n);
/// (1 − CV)^{-1}(2 − C)(2 − C − NU)^{-1}
n := 0
/// (1 − CV)^{-1}(2 − C)(2 − C − U)^{-1}
```

Fig. 2. Program equivalence follows from the equality of the resulting SOP (Lemma 3).

are equivalent (i.e., $[\![L]\!] = [\![I]\!]$) by showing that $[\![\texttt{if}\ (\mathtt{n} > 0)\ \{P \fatsemi I\}]\!] = [\![I]\!]$ as suggested by Lemma 2. The latter is achieved as in Lemma 3: We run both programs on the input SOP $g_{N,C} = (1 - NU)^{-1}(1 - CV)^{-1}$, where U, V are meta indeterminates corresponding to N and C, respectively, and check if the results are equal. Note that I is the loop-free specification from Example 1; thus by transitivity, the loop L is equivalent to the loop in Fig. 1. ◁

6 Case Studies

We have implemented our techniques in Python as a prototype called Prodigy[3]: PRObability DIstributions via GeneratingfunctionologY. By interfacing with different computer algebra systems (CAS), e.g., `Sympy` [49] and `GiNaC` [10,57] – as backends for symbolic computation of PGF and SOP semantics – Prodigy decides whether a given probabilistic loop agrees with an (invariant) specification encoded as a loop-free `ReDiP` program. Furthermore, it supports efficient queries on various quantities associated with the output distribution.

In what follows, we demonstrate in particular the applicability of our techniques to programs featuring stochastic dependency, parametrization, and nested loops. The examples are all presented in the same way: the iterative program on the left side and its corresponding specification on the right. The presented programs are all UAST, given the parameters are instantiated from a suitable value domain.[4] For each example, we report the time for performing the equivalence check on a 2,4 GHz Intel i5 Quad-Core processor with 16GB RAM running macOS Monterey 12.0.1. Additional examples can be found in [18, Appx. E].

```
while (c > 0) {                          if (c > 0) {
    { n := n + 1 } [1/2] { m := m + 1 };     tmp := binomial(1/2, c);
    c := c − 1;                              m += tmp; n += c − tmp;
    tmp := 0                                 c := 0;
}                                            tmp := 0 }
```

Fig. 3. Generating complementary binomial distributions (for `n`, `m`) by coin flips. `binomial`(1/2, `c`) is an alias for `iid`(`bernoulli`(1/2), `c`).

```
while (c = 1 ∧ t ≤ 1) {                  if (c = 1 ∧ t ≤ 1) {
    if (t = 0) {                             c := 0
        { c := 0 } [a] { t := 1 }            if (t = 0) {
    } else {                                     t := bernoulli((1−a)b/(a+b−ab));
        { c := 0 } [b] { t := 0 }            } else {
    }                                            t := bernoulli(b/(a+b−ab));
}                                            } }
```

Fig. 4. A program modeling two dueling cowboys with parametric hit probabilities.

Example 3 (Complementary Binomial Distributions). We show that the program in Fig. 3 generates a joint distribution on `n`, `m` such that both `n` and `m` are binomially distributed with support `c` and are complementary in the sense that `n` + `m` = `c` holds certainly (if `n` = `m` = 0 initially, otherwise the variables

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

[4] Parameters of Example 4 have to be instantiated with a probability value in (0, 1).

are incremented by the corresponding amounts). PRODIGY automatically checks that the loop agrees with the specification in 18.3 ms. The resulting distribution can then be analyzed for any given input PGF g by computing $[\![I]\!](g)$, where I is the loop-free program. For example, for input $g = C^{10}$, the distribution as computed by PRODIGY has the *factorized* closed form $(\frac{M+N}{2})^{10}$. The CAS backends exploit such factorized forms to perform algebraic manipulations more efficiently compared to fully expanded forms. For instance, we can evaluate the queries $E[m^3+2mn+n^2] = 235$, or $Pr(m > 7 \wedge n < 3) = 7/128$, almost instantly. ◁

Example 4 (Dueling Cowboys [46]). The program in Fig. 4 models a duel of two cowboys with *parametric* hit probabilities `a` and `b`. Variable `t` indicates the cowboy who is currently taking his shot, and `c` monitors the state of the duel (`c` = 1: duel is still running, `c` = 0: duel is over). PRODIGY automatically verifies the specification in 11.97 ms. We defer related problems – e.g., *synthesizing* parameter values to meet a parameter-free specification – to future work. ◁

```
while (x > 0) {
    y := 1⁄2;
    while (y = 1) {
        { y := 0 } [1/2] { x := x + 1 } };
    x := x − 1⁄2;
    c += 1⁄2;
}
```

```
/* inner invariant */
if (y = 1) {
    x += geometric(1/2)⁄2;
    y := 0⁄2;
}
```

```
/* outer invariant */
if (x > 0) {
    c := iid(catalan(1/2), x)⁄2;
    x := 0⁄2;
    y := 0
}
```

Fig. 5. Nested loops with invariants for the inner and outer loop.

Example 5 (Nested Loops). The inner loop of the program in Fig. 5 modifies `x` which influences the termination behavior of the outer loop. Intuitively, the program models a random walk on $\mathbb{N}$: In every step, the value of the current position `x` changes by some random $\delta \in \{-1, 0, 1, 2, \ldots\}$ such that $\delta + 1$ is geometrically distributed. The example demonstrates how our technique enables *compositional* reasoning. We first provide a loop-free specification for the inner loop, prove its correctness, and then simply *replace* the inner loop by its specification, yielding a program without nested loops. This feature is a key benefit of reusing the loop-free fragment of `ReDiP` as a specification language. Moreover, existing techniques that cannot handle nested loops can profit from it; in fact, we can prove the overall program to be UAST using the rule of [47]. Interestingly, the outer loop has *infinite expected runtime* (for any input distribution where the probability that $\mathtt{x} > 0$ is positive). We can prove this by *querying the expected value* of the program variable `c` in the resulting output distribution. The automatically computed result is ∞, which indeed proves that the expected runtime of this program is not finite. This example furthermore shows that our technique can be generalized beyond rational functions since the PGF of the `catalan`(p) distribution is $(1 - \sqrt{1 - 4p(1-p)T}) \,/\, 2p$, i.e., algebraic but not rational. We leave

a formal generalization of the decidability result from Theorem 4 to algebraic functions for future work. PRODIGY verifies this example in 29.17ms. ◁

Scalability Issue. It is not difficult to construct programs where PRODIGY poorly scales: its performance depends highly on the number of consecutive probabilistic branches and the size of the constant n in guards (requiring n-th order PGF derivation, cf. Table 2).

7 Related Work

This section surveys research efforts that are highly related to our approach in terms of semantics, inference, and equivalence checking of probabilistic programs.

Forward Semantics of Probabilistic Programs. Kozen established in his seminal work [43] a generic way of giving forward, denotational semantics to probabilistic programs as *distribution transformers*. Klinkenberg et al. [42] instantiated Kozen's semantics as PGF transformers. We refine the PGF semantics substantially such that it enjoys the following crucial properties: (i) our PGF transformers (when restricted to loop-free `ReDiP` programs) preserve closed-form PGF and thus are effectively constructable. In contrast, the existing PGF semantics in [42] operates on infinite sums in a non-constructive fashion; (ii) our PGF semantics naturally extends to SOP, which serves as the key to reason about the exact behavior of unbounded loops (under possibly uncountably many inputs) in a fully automatic manner. The PGF semantics in [42], however, supports only (over-)approximations of looping behaviors and can hardly be automated; and (iii) our PGF semantics is capable of interpreting program constructs like i.i.d. sampling that is of particular interest in practice.

Backward Semantics of Probabilistic Programs. Many verification systems for probabilistic programs make use of backward, denotational semantics – most pertinently, the *weakest preexpectation* (WP) calculi [38,46] as a quantitative extension of Dijkstra's weakest preconditions [19]. The WP of a probabilistic program C w.r.t. a postexpectation g, denoted by $\mathsf{wp}[\![C]\!](g)(\cdot)$, maps every initial program state σ to the expected value of g evaluated in final states reached after executing C on σ. In contrast to Dijkstra's predicate transformer semantics which admits also strongest postconditions, the counterpart of "strongest post-expectations" does unfortunately not exist [36, Chap. 7], thereby not amenable to forward reasoning. We remark, in particular, that checking program equivalence via WP is difficult, if not impossible, since it amounts to reasoning about uncountably many postexpectations g. We refer interested readers to [5, Chaps. 1–4] for more recent advancements in formal semantics of probabilistic programs.

Probabilistic Inference. There are a handful of probabilistic systems that employ an alternative forward semantics based on *probability density function* (PDF) representations of distributions, e.g., (λ)PSI [24,25], AQUA [32], Hakaru [14,52],

and the density compiler in [11,12]. These systems are dedicated to probabilistic inference for programs encoding continuous distributions (or joint discrete-continuous distributions). Reasoning about the underlying PDF representations, however, amounts to resolving complex integral expressions in order to answer inference queries, thus confining these techniques either to (semi-)numerical methods [11,12,14,32,52] or exact methods yet limited to bounded looping behaviors [24,25]. Apart from these inference systems, a recently developed language called Dice [31] featuring exact inference for discrete probabilistic programs is also confined to statically bounded loops. The tool Mora [7,8] 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.

Equivalence of Probabilistic Programs. Murawski and Ouaknine [51] showed an EXPTIME decidability result for checking the equivalence of probabilistic programs over *finite* data types by recasting the problem in terms of probabilistic finite automata [23,41,56]. Their techniques have been automated in the equivalence checker APEX [45]. Barthe et al. [4] proved a 2-EXPTIME decidability result for checking equivalence of *straight-line* probabilistic programs (with deterministic inputs and no loops nor recursion) interpreted over all possible extensions of a finite field. Barthe et al. [3] developed a relational Hoare logic for probabilistic programs, which has been extensively used for, amongst others, proving program equivalence with applications in provable security and side-channel analysis.

The decidability result established in this paper is *orthogonal* to the aforementioned results: (i) our decidability for checking $L \sim S$ applies to discrete probabilistic programs L with *unbounded* looping behaviors over a possibly *infinite* state space; the specification S – though, admitting no loops – encodes a possibly *infinite-support* distribution; yet as a compromise, (ii) our decidability result is confined to `ReDiP` programs that necessarily terminate almost-surely on all inputs, and involve only distributions with rational closed-form PGF.

8 Conclusion and Future Work

We showed the decidability of – and have presented a fully-automated technique to verifying – whether a (possibly unbounded) probabilistic loop is equivalent to a loop-free specification program. Future directions include determining the complexity of our decision problem; amending the method to continuous distributions using, e.g., *characteristic functions*; extending the notion of probabilistic equivalence to probabilistic refinements; exploring PGF-based counterexample-guided synthesis of quantitative loop invariants (see [18, Appx. F.6] for generating counterexamples); and tackling Bayesian inference.

Acknowledgments. The authors thank Philipp Schröer for providing support for his tool PROBABLY (https://github.com/Philipp15b/Probably) which forms the basis of our implementation.

References

1. Arvo, J., Kirk, D.B.: Particle transport and image synthesis. In: SIGGRAPH, pp. 63–66. ACM (1990)
2. Aspnes, J., Herlihy, M.: Fast randomized consensus using shared memory. J. Algorithms **11**(3), 441–461 (1990)
3. Barthe, G., Grégoire, B., Béguelin, S.Z.: Formal certification of code-based cryptographic proofs. In: POPL, pp. 90–101. ACM (2009)
4. Barthe, G., Jacomme, C., Kremer, S.: Universal equivalence and majority of probabilistic programs over finite fields. In: LICS, pp. 155–166. ACM (2020)
5. Barthe, G., Katoen, J., Silva, A. (eds.): Foundations of Probabilistic Programming. Cambridge University Press, Cambridge (2020)
6. Barthe, G., Köpf, B., Olmedo, F., Béguelin, S.Z.: Probabilistic relational reasoning for differential privacy. ACM Trans. Program. Lang. Syst. **35**(3), 9:1–9:49 (2013)
7. Bartocci, E., Kovács, L., Stankovič, M.: Analysis of Bayesian networks via prob-solvable loops. In: Pun, V.K.I., Stolz, V., Simao, A. (eds.) ICTAC 2020. LNCS, vol. 12545, pp. 221–241. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-64276-1_12
8. Bartocci, E., Kovács, L., Stankovič, M.: Mora - automatic generation of moment-based invariants. In: TACAS 2020. LNCS, vol. 12078, pp. 492–498. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45190-5_28
9. Batz, K., Chen, M., Kaminski, B.L., Katoen, J.-P., Matheja, C., Schröer, P.: Latticed k-induction with an application to probabilistic programs. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12760, pp. 524–549. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81688-9_25
10. Bauer, C., Frink, A., Kreckel, R.: Introduction to the GiNaC framework for symbolic computation within the C++ programming language. J. Symb. Comput. **33**(1), 1–12 (2002)
11. Bhat, S., Agarwal, A., Vuduc, R.W., Gray, A.G.: A type theory for probability density functions. In: POPL, pp. 545–556. ACM (2012)
12. Bhat, S., Borgström, J., Gordon, A.D., Russo, C.V.: Deriving probability density functions from probabilistic functional programs. Log. Methods Comput. Sci. **13**(2) (2017)
13. Carbin, M., Misailovic, S., Rinard, M.C.: Verifying quantitative reliability for programs that execute on unreliable hardware. Commun. ACM **59**(8), 83–91 (2016)
14. Carette, J., Shan, C.-C.: Simplifying probabilistic programs using computer algebra. In: Gavanelli, M., Reppy, J. (eds.) PADL 2016. LNCS, vol. 9585, pp. 135–152. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-28228-2_9
15. Chakarov, A., Sankaranarayanan, S.: Probabilistic program analysis with martingales. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 511–526. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_34
16. Chatterjee, K., Fu, H., Goharshady, A.K.: Termination analysis of probabilistic programs through Positivstellensatz's. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 3–22. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_1
17. Chatterjee, K., Fu, H., Novotný, P.: Termination analysis of probabilistic programs with martingales, pp. 221–258. In: Barthe et al. [5] (2020)
18. Chen, M., Katoen, J., Klinkenberg, L., Winkler, T.: Does a program yield the right distribution? Verifying probabilistic programs via generating functions. CoRR abs/2205.01449 (2022)

19. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM **18**(8), 453–457 (1975)
20. Evans, O., Stuhlmüller, A., Salvatier, J., Filan, D.: Modeling agents with probabilistic programs. http://agentmodels.org (2017). Accessed 17 Jan 2022
21. Flajolet, P., Pelletier, M., Soria, M.: On Buffon machines and numbers. In: SODA, pp. 172–183. SIAM (2011)
22. Flajolet, P., Sedgewick, R.: Analytic Combinatorics. Cambridge University Press, Cambridge (2009)
23. Forejt, V., Jancar, P., Kiefer, S., Worrell, J.: Language equivalence of probabilistic pushdown automata. Inf. Comput. **237**, 1–11 (2014)
24. Gehr, T., Misailovic, S., Vechev, M.: PSI: exact symbolic inference for probabilistic programs. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 62–83. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_4
25. Gehr, T., Steffen, S., Vechev, M.T.: λPSI: exact inference for higher-order probabilistic programs. In: PLDI, pp. 883–897. ACM (2020)
26. Gordon, A.D., Henzinger, T.A., Nori, A.V., Rajamani, S.K.: Probabilistic programming. In: FOSE, pp. 167–181. ACM (2014)
27. Hammersley, J.: Monte Carlo Methods. Springer Science & Business Media (2013)
28. Hark, M., Kaminski, B.L., Giesl, J., Katoen, J.: Aiming low is harder: induction for lower bounds in probabilistic program verification. Proc. ACM Program. Lang. **4**(POPL), 37:1–37:28 (2020)
29. Heninger, N.: RSA, DH and DSA in the wild. In: Bos, J., Stam, M. (eds.) Computational Cryptography: Algorithmic Aspects of Cryptology, pp. 140–181. Cambridge University Press, Cambridge (2021)
30. Hicks, M.: What is probabilistic programming? In: The Programming Languages Enthusiast (2014). http://www.pl-enthusiast.net/2014/09/08. Accessed 09 Dec 2021
31. Holtzen, S., den Broeck, G.V., Millstein, T.D.: Scaling exact inference for discrete probabilistic programs. Proc. ACM Program. Lang. **4**(OOPSLA), 140:1–140:31 (2020)
32. Huang, Z., Dutta, S., Misailovic, S.: AQUA: automated quantized inference for probabilistic programs. In: Hou, Z., Ganesh, V. (eds.) ATVA 2021. LNCS, vol. 12971, pp. 229–246. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-88885-5_16
33. Jacobs, B., Zanasi, F.: The logical essentials of Bayesian reasoning, pp. 295–331. In: Barthe et al. [5] (2020)
34. Jansen, N., Dehnert, C., Kaminski, B.L., Katoen, J.-P., Westhofen, L.: Bounded model checking for probabilistic programs. In: Artho, C., Legay, A., Peled, D. (eds.) ATVA 2016. LNCS, vol. 9938, pp. 68–85. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46520-3_5
35. Johnson, N., Kotz, S., Kemp, A.: Univariate Discrete Distributions. Wiley, Hoboken (1993)
36. Jones, C.: Probabilistic non-determinism. Ph.D. thesis, University of Edinburgh, UK (1990)
37. Kajiya, J.T.: The rendering equation. In: SIGGRAPH, pp. 143–150. ACM (1986)
38. Kaminski, B.L.: Advanced weakest precondition calculi for probabilistic programs. Ph.D. thesis, RWTH Aachen University, Germany (2019)
39. Kaminski, B.L., Katoen, J.-P., Matheja, C.: On the hardness of analyzing probabilistic programs. Acta Informatica **56**(3), 255–285 (2018). https://doi.org/10.1007/s00236-018-0321-1

40. Kaminski, B.L., Katoen, J., Matheja, C., Olmedo, F.: Weakest precondition reasoning for expected runtimes of randomized algorithms. J. ACM **65**(5), 30:1–30:68 (2018)
41. Kiefer, S., Murawski, A.S., Ouaknine, J., Wachter, B., Worrell, J.: Language equivalence for probabilistic automata. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 526–540. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_42
42. Klinkenberg, L., Batz, K., Kaminski, B.L., Katoen, J.-P., Moerman, J., Winkler, T.: Generating functions for probabilistic programs. In: Fernández, M. (ed.) LOPSTR 2020. LNCS, vol. 12561, pp. 231–248. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-68446-4_12
43. Kozen, D.: Semantics of probabilistic programs. J. Comput. Syst. Sci. **22**(3), 328–350 (1981)
44. Kozen, D.: A probabilistic PDL. J. Comput. Syst. Sci. **30**(2), 162–178 (1985)
45. Legay, A., Murawski, A.S., Ouaknine, J., Worrell, J.: on automated verification of probabilistic programs. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 173–187. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_13
46. McIver, A., Morgan, C.: Abstraction, Refinement and Proof For Probabilistic Systems. Monographs in Computer Science. Springer, New York (2005). https://doi.org/10.1007/b138392
47. McIver, A., Morgan, C., Kaminski, B.L., Katoen, J.: A new proof rule for almost-sure termination. PACMPL **2**(POPL), 33:1–33:28 (2018)
48. van de Meent, J., Paige, B., Yang, H., Wood, F.: An introduction to probabilistic programming. CoRR abs/1809.10756 (2018)
49. Meurer, A., et al.: SymPy: symbolic computing in Python. PeerJ Comput. Sci. **3**, e103 (2017)
50. Moosbrugger, M., Bartocci, E., Katoen, J.-P., Kovács, L.: The probabilistic termination tool amber. In: Huisman, M., Păsăreanu, C., Zhan, N. (eds.) FM 2021. LNCS, vol. 13047, pp. 667–675. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-90870-6_36
51. Murawski, A.S., Ouaknine, J.: On probabilistic program equivalence and refinement. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 156–170. Springer, Heidelberg (2005). https://doi.org/10.1007/11539452_15
52. Narayanan, P., Carette, J., Romano, W., Shan, C., Zinkov, R.: Probabilistic inference by program transformation in Hakaru (system description). In: Kiselyov, O., King, A. (eds.) FLOPS 2016. LNCS, vol. 9613, pp. 62–79. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-29604-3_5
53. Schneider, M.: Self-stabilization. ACM Comput. Surv. **25**(1), 45–67 (1993)
54. Shamsi, S.M., Farina, G.P., Gaboardi, M., Napp, N.: Probabilistic programming languages for modeling autonomous systems. In: MFI, pp. 32–39. IEEE (2020)
55. Tijms, H.C.: A First Course in Stochastic Models. Wiley, Hoboken (2003)
56. Tzeng, W.: A polynomial-time algorithm for the equivalence of probabilistic automata. SIAM J. Comput. **21**(2), 216–227 (1992)
57. Vollinga, J.: GiNaC-symbolic Computation with C++. Nucl. Instrum. Methods Phys. Res. **559**(1), 282–284 (2006)
58. Wang, D., Hoffmann, J., Reps, T.W.: Central moment analysis for cost accumulators in probabilistic programs. In: PLDI, pp. 559–573. ACM (2021)
59. Wang, J., Sun, Y., Fu, H., Chatterjee, K., Goharshady, A.K.: Quantitative analysis of assertion violations in probabilistic programs. In: PLDI, pp. 1171–1186. ACM (2021)

60. Wilf, H.S.: Generating Functionology. CRC Press, Boca Raton (2005)
61. Ying, M.: Floyd-Hoare logic for quantum programs. ACM Trans. Program. Lang. Syst. **33**(6), 19:1–19:49 (2011)