Generating Functions for Probabilistic Programs

Lutz Klinkenberg[1(✉)], Kevin Batz[1], Benjamin Lucien Kaminski[1,2], Joost-Pieter Katoen[1], Joshua Moerman[1], and Tobias Winkler[1]

[1] RWTH Aachen University, 52062 Aachen, Germany
{lutz.klinkenberg,kevin.batz,katoen,joshua,
tobias.winkler}@cs.rwth-aachen.de
[2] University College London, London, UK
b.kaminski@ucl.ac.uk

Abstract. This paper investigates the usage of generating functions (GFs) encoding measures over the program variables for reasoning about discrete probabilistic programs. To that end, we define a denotational GF-transformer semantics for probabilistic while-programs, and show that it instantiates Kozen's seminal distribution transformer semantics. We then study the effective usage of GFs for program analysis. We show that finitely expressible GFs enable checking super-invariants by means of computer algebra tools, and that they can be used to determine termination probabilities. The paper concludes by characterizing a class of—possibly infinite-state—programs whose semantics is a rational GF encoding a discrete phase-type distribution.

Keywords: Probabilistic programs · Quantitative verification · Semantics · Formal power series

1 Introduction

Probabilistic programs are sequential programs for which coin flipping is a first-class citizen. They are used e.g. to represent randomized algorithms, probabilistic graphical models such as Bayesian networks, cognitive models, or security protocols. Although probabilistic programs are typically rather small, their analysis is intricate. For instance, approximating expected values of program variables at program termination is as hard as the universal halting problem [22]. Determining higher moments such as variances is even harder. Deductive program verification techniques based on a quantitative version of weakest preconditions [20,25] enable to reason about the outcomes of probabilistic programs, such as what is the probability that a program variable equals a certain value. Dedicated analysis techniques have been developed to e.g., determine tail bounds [5], decide almost-sure termination [7,26], or to compare programs [1].

This research was funded by the ERC AdG project FRAPPANT (787914) and the DFG RTG 2236 UnRAVeL.

M. Fernández (Ed.): LOPSTR 2020, LNCS 12561, pp. 231–248, 2021.
https://doi.org/10.1007/978-3-030-68446-4_12

This paper aims at exploiting the well-tried potential of *probability generating functions* (PGFs) [19] for analyzing probabilistic programs. In our setting, PGFs are power series representations encoding *discrete* probability mass functions of joint distributions over program variables. PGF representations — in particular if finite—enable a simple extraction of important information from the encoded distributions such as expected values, higher moments, termination probabilities or stochastic independence of program variables.

To enable the usage of PGFs for program analysis, we define a denotational semantics of a simple probabilistic while-language akin to probabilistic GCL [25]. Our semantics is defined in a *forward* manner: given an input distribution over program variables as a PGF, it yields a PGF representing the resulting subdistribution. The "missing" probability mass represents the probability of non-termination. More accurately, our denotational semantics transforms *formal power series* (FPS). Those form a richer class than PGFs, which allows for overapproximations of probability distributions. The meaning of while-loops are given as least fixed points of FPS transformers. It is shown that our semantics is in fact an instantiation of Kozen's seminal distribution-transformer semantics [23].

The semantics provides a sound basis for program analysis using PGFs. Using Park's Lemma, we obtain a simple technique to prove whether a given FPS overapproximates a program's semantics i.e., whether an FPS is a so-called super-invariant. Such upper bounds can be quite useful: for almost-surely terminating programs, such bounds can provide *exact* program semantics, whereas, if the mass of an overapproximation is strictly less than one, the program is *provably non-almost-surely terminating*. This result is illustrated on a non-trivial random walk and on examples illustrating that checking whether an FPS is a super-invariant can be *automated* using computer algebra tools.

In addition, we characterize a class of—possibly infinite-state—programs whose PGF semantics is a rational function. These *homogeneous bounded programs* (HB programs) are characterized by loops in which each unbounded variable has no effect on the loop guard and is in each loop iteration incremented by a quantity independent of its own value. Operationally speaking, HB programs can be considered as finite-state Markov chains with rewards that can grow unboundedly large. It is shown that the rational PGF of any program that is equivalent to an almost-surely terminating HB program represents a multi-variate discrete phase-type distribution [29]. We illustrate this result by obtaining a closed-form characterization for the well-studied infinite-state dueling cowboys example [25,32].

Related Work. This paper presents a denotational semantics of probabilistic programs using PGFs and shows how the PGF representation can be exploited for program analysis. Our PGF semantics is defined in a forward manner: starting from an initial distribution on inputs, it determines the exact probability distribution over the program variables on termination. This fits within the realm of Kozen's denotational semantics [23]. Di Pierro and Wiklicky [10] provided a forward, denotational semantics of a similar programming language using infinite-dimensional

Hilbert spaces to provide a basis for program analysis by means of probabilistic abstract interpretation. Other semantics include backward denotational semantics using weakest preconditions [25] and operational semantics, e.g., using Markov chains [13].

Whereas advanced simulation techniques are the primary analysis technique for modern probabilistic programming languages, our approach using PGFs is exact. Our PGF approach is a forward approach and yields full probability distributions for a given program input. This is similar in spirit as in EfProb [8], a calculus based on a categorical semantics to reason about loop-free programs with discrete, continuous and quantum probability. Wp-reasoning [25] is an alternative analysis technique to prove properties of probabilistic programs. It determines the weakest pre-expectation function—the quantitative analogue of preconditions in classical program verification—in a backward manner for a given post-expectation, the property to be proven. Related program analysis techniques include the usage of couplings to prove program equivalence [1], abstract interpretation [9] and Hoare logics [15].

To the best of our knowledge, PGFs have recent scant attention in the analysis of probabilistic programs. A notable exception is [4] in which generating functions of finite Markov chains are obtained by Padé approximation. Computer algebra systems have been used to transform probabilistic programs [6], and more recently in the automated generation of moment-based loop invariants [2].

Organization of this Paper. After recapping FPSs and PGFs in Sects. 2–3, we define our FPS transformer semantics in Sect. 4, discuss some elementary properties and show it instantiates Kozen's distribution transformer semantics [23]. Section 5 presents our approach for verifying upper bounds to loop invariants and illustrates this by various non-trivial examples. In addition, it characterizes programs that are representable as finite-state Markov chains equipped with rewards and presents the relation to discrete phase-type distributions. Section 6 concludes the paper. The full paper can be found on ArXiv.[1]

2 Formal Power Series

Our goal is to make the potential of probability generating functions available to the formal verification of probabilistic programs. The programs we consider will, without loss of generality, operate on a fixed set of k program variables. The valuations of those variables range over $\mathbb{N}$. A *program state* σ is hence a vector in $\mathbb{N}^k$. We denote the state $(0,\ldots,0)$ by $\mathbf{0}$.

A prerequisite for understanding probability generating functions are (multivariate) *formal power series—a special way of representing a potentially infinite k-dimensional array.* For k=1, this amounts to representing a *sequence.*

[1] https://arxiv.org/abs/2007.06327.

Definition 1 (Formal Power Series). *Let* $\mathbf{X} = X_1, \ldots, X_k$ *be a fixed sequence of* k *distinct formal indeterminates. For a state* $\sigma = (\sigma_1, \ldots, \sigma_k) \in \mathbb{N}^k$*, let* $\mathbf{X}^\sigma$ *abbreviate the formal multiplication* $X_1^{\sigma_1} \cdots X_k^{\sigma_k}$*. The latter object is called a* monomial *and we denote the set of all monomials over* $\mathbf{X}$ *by* $\mathrm{Mon}(\mathbf{X})$*. A (multivariate)* formal power series (FPS) *is a formal sum*

$$F = \sum_{\sigma \in \mathbb{N}^k} [\sigma]_F \cdot \mathbf{X}^\sigma \,, \qquad \text{where} \qquad [\,\cdot\,]_F : \quad \mathbb{N}^k \to \mathbb{R}^\infty_{\geq 0} \,,$$

where $\mathbb{R}^\infty_{\geq 0}$ *denotes the extended positive real line. We denote the set of all FPSs by* FPS*. Let* $F, G \in \mathsf{FPS}$*. If* $[\sigma]_F < \infty$ *for all* $\sigma \in \mathbb{N}^k$*, we denote this fact by* $F \ll \infty$*. The* addition $F + G$ *and* scaling $r \cdot F$ *by a scalar* $r \in \mathbb{R}^\infty_{\geq 0}$ *is defined coefficient-wise by*

$$F + G = \sum_{\sigma \in \mathbb{N}^k} \left([\sigma]_F + [\sigma]_G\right) \cdot \mathbf{X}^\sigma \qquad \text{and} \qquad r \cdot F = \sum_{\sigma \in \mathbb{N}^k} r \cdot [\sigma]_F \cdot \mathbf{X}^\sigma \,.$$

For states $\sigma = (\sigma_1, \ldots, \sigma_k)$ *and* $\tau = (\tau_1, \ldots, \tau_k)$*, we define* $\sigma + \tau = (\sigma_1 + \tau_1, \ldots, \sigma_k + \tau_k)$*. The* multiplication $F \cdot G$ *is given as their Cauchy product (or discrete convolution)*

$$F \cdot G = \sum_{\sigma, \tau \in \mathbb{N}^k} [\sigma]_F \cdot [\tau]_G \cdot \mathbf{X}^{\sigma + \tau} \,.$$

Drawing coefficients from the extended reals enables us to define a *complete lattice* on FPSs in Sect. 4. Our analyses in Sect. 5 will, however, only consider FPSs with $F \ll \infty$.

3 Generating Functions

> A generating function is a device somewhat similar to a bag. Instead of carrying many little objects detachedly, which could be embarrassing, we put them all in a bag, and then we have only one object to carry, the bag.
>
> — George Pólya [31]

Formal power series pose merely a particular way of encoding an infinite k-dimensional array as yet another infinitary object, but we still carry all objects forming the array (the coefficients of the FPS) detachedly and there seems to be no advantage in this particular encoding. It even seems more bulky. We will now, however, see that this bulky encoding can be turned into a one-object bag carrying all our objects: the *generating function*.

Definition 2 (Generating Functions). *The* generating function *of a formal power series* $F = \sum_{\sigma \in \mathbb{N}^k} [\sigma]_F \cdot \mathbf{X}^\sigma \in \mathsf{FPS}$ *with* $F \ll \infty$ *is the* partial *function*

$$f : \quad [0, 1]^k \dashrightarrow \mathbb{R}_{\geq 0}, \quad (x_1, \ldots, x_k) \mapsto \sum_{\sigma = (\sigma_1, \ldots, \sigma_k) \in \mathbb{N}^k} [\sigma]_F \cdot x_1^{\sigma_1} \cdots x_k^{\sigma_k} \,.$$

In other words: in order to turn an FPS into its generating function, we merely treat every *formal* indeterminate X_i as an *"actual"* indeterminate x_i, and the formal multiplications and the formal sum also as *"actual"* ones. The generating function f of F is *uniquely determined* by F as we require all coefficients of F to be non-negative, and so the ordering of the summands is irrelevant: For a given point $\boldsymbol{x} \in [0, 1]^k$, the sum defining $f(\boldsymbol{x})$ either converges *absolutely* to some positive real or diverges absolutely to ∞. In the latter case, f is undefined at $\boldsymbol{x}$ and hence f may indeed be partial.

Since generating functions stem from formal power series, they are infinitely often differentiable at $\mathbf{0} = (0, \ldots, 0)$. Because of that, we can recover F from f as the (multivariate) Taylor expansion of f at $\mathbf{0}$.

Definition 3 (Multivariate Derivatives and Taylor Expansions). *For $\sigma = (\sigma_1, \ldots, \sigma_k) \in \mathbb{N}^k$, we write $f^{(\sigma)}$ for the function f differentiated σ_1 times in x_1, σ_2 times in x_2, and so on. If f is infinitely often differentiable at $\mathbf{0}$, then the* Taylor expansion of f at $\mathbf{0}$ *is given by*

$$\sum_{\sigma \in \mathbb{N}^k} \frac{f^{(\sigma)}(\mathbf{0})}{\sigma_1! \cdots \sigma_k!} \cdot x_1^{\sigma_1} \cdots x_k^{\sigma_k} \ .$$

If we replace every indeterminate x_i by the *formal* indeterminate X_i in the Taylor expansion of generating function f of F, then we obtain the formal power series F. It is in precisely that sense, that f *generates* F.

Example 1 (Formal Power Series and Generating Functions). Consider the infinite (1-dimensional) sequence $1/2$, $1/4$, $1/8$, $1/16$, ... Its (univariate) FPS—the entity carrying all coefficients detachedly—is given as

$$\frac{1}{2} + \frac{1}{4}X + \frac{1}{8}X^2 + \frac{1}{16}X^3 + \frac{1}{32}X^4 + \frac{1}{64}X^5 + \frac{1}{128}X^6 + \frac{1}{256}X^7 + \ldots \ . \qquad (\dagger)$$

On the other hand, its generating function—the bag—is given concisely by

$$\frac{1}{2 - x} \ . \qquad (\flat)$$

Figuratively speaking, (†) is itself the infinite sequence $a_n \coloneqq \frac{1}{2^n}$, whereas (♭) is a bag with the label "infinite sequence $a_n \coloneqq \frac{1}{2^n}$". The fact that (†) generates (♭), follows from the Taylor expansion of $\frac{1}{2-x}$ at 0 being $\frac{1}{2} + \frac{1}{4}x + \frac{1}{8}x^2 + \ldots$. △

The potential of generating functions is that manipulations to the functions—i.e. to the concise representations—are in a one-to-one correspondence to the associated manipulations to FPSs [12]. For instance, if $f(x)$ is the generating function of F encoding the sequence a_1, a_2, a_3, ..., then the function $f(x) \cdot x$ is the generating function of $F \cdot X$ which encodes the sequence 0, a_1, a_2, a_3, ...

As another example for correspondence between operations on FPSs and generating functions, if $f(x)$ and $g(x)$ are the generating functions of F and G, respectively, then $f(x) + g(x)$ is the generating function of $F + G$.

Example 2 (Manipulating Generating Functions). Revisiting Example 1, if we multiply $\frac{1}{2-x}$ by x, we change the label on our bag from "infinite sequence $a_n \coloneqq \frac{1}{2^n}$" to "a 0 followed by an infinite sequence $a_{n+1} \coloneqq \frac{1}{2^n}$" and—just by changing the label—the bag will now contain what it says on its label. Indeed, the Taylor expansion of $\frac{x}{2-x}$ at 0 is $0 + \frac{1}{2}x + \frac{1}{4}x^2 + \frac{1}{8}x^3 + \frac{1}{16}x^4 + \ldots$ encoding the sequence 0, 1/2, 1/4, 1/8, 1/16, ... △

Due to the close correspondence of FPSs and generating functions [12], we use both concepts interchangeably, as is common in most mathematical literature. We mostly use FPSs for definitions and semantics, and generating functions in calculations and examples.

Probability Generating Functions. We now use formal power series to represent probability distributions.

Definition 4 (Probability Subdistribution). *A* probability subdistribution *(or simply subdistribution) over* $\mathbb{N}^k$ *is a function*

$$\mu\colon \quad \mathbb{N}^k \;\to\; [0,1], \qquad \textit{such that} \qquad |\mu| \;=\; \sum_{\sigma\in\mathbb{N}^k} \mu(\sigma) \leq 1 \;.$$

We call $|\mu|$ *the* mass of μ. *We say that* μ *is a* (full) distribution *if* $|\mu| = 1$, *and a* proper subdistribution *if* $|\mu| < 1$. *The set of all subdistributions on* $\mathbb{N}^k$ *is denoted by* $\mathcal{D}_{\leq}(\mathbb{N}^k)$ *and the set of all full distributions by* $\mathcal{D}(\mathbb{N}^k)$.

We need subdistributions for capturing non-termination. The "missing" probability mass $1 - |\mu|$ precisely models the probability of non-termination.

The generating function of a (sub-)distribution is called a *probability generating function*. Many properties of a distribution μ can be read off from its generating function G_μ in a simple way. We demonstrate how to extract a few common properties in the following example.

Example 3 (Geometric Distribution PGF). Recall Example 1. The presented formal power series encodes a *geometric distribution* μ_{geo} with parameter 1/2 of a single variable X. The fact that μ_{geo} is a proper probability distribution, for instance, can easily be verified computing $G_{geo}(1) = \frac{1}{2-1} = 1$. The expected value of X is given by $G'_{geo}(1) = \frac{1}{(2-1)^2} = 1$. △

Extracting Common Properties. Important information about probability distributions is, for instance, the first and higher moments. In general, the k^{th} factorial moment of variable X_i can be extracted from a PGF by computing $\frac{\partial^k G}{\partial X_i^k}(1,\ldots,1)$.[2] This includes the mass $|G|$ as the 0^{th} moment. The marginal distribution of variable X_i can simply be extracted from G by $G(1,\ldots,X_i,\ldots,1)$. We also note that PGFs can treat *stochastic independence*. For instance, for a bivariate PGF H we can check for stochastic independence of the variables X and Y by checking whether $H(X,Y) = H(X,1)\cdot H(1,Y)$.

[2] In general, one must take the limit $X_i \to 1$ from below.

4 FPS Semantics for pGCL

In this section, we give denotational semantics to probabilistic programs in terms of FPS transformers and establish some elementary properties useful for program analysis. We begin by endowing FPSs and PGFs with an order structure:

Definition 5 (Order on FPS). *For all $F, G \in \mathsf{FPS}$, let*

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

Lemma 1 (Completeness of $\preceq$ on FPS). *$(\mathsf{FPS}, \preceq)$ is a complete latttice.*

4.1 FPS Transformer Semantics

Recall that we assume programs to range over exactly k variables with valuations in $\mathbb{N}^k$. Our program syntax is similar to Kozen [23] and McIver & Morgan [25].

Definition 6 (Syntax of pGCL [23,25]). *A program P in* probabilistic Guarded Command Language *(pGCL) adheres to the grammar*

$$\begin{aligned} P ::= \;& \texttt{skip} \mid \mathtt{x}_i := E \mid P; P \mid \{P\}\ [p]\ \{P\} \\ & \mid \texttt{if}(B)\ \{P\}\ \texttt{else}\ \{P\} \mid \texttt{while}\,(B)\ \{P\}\ , \end{aligned}$$

where $\mathtt{x}_i \in \{\mathtt{x}_1, \ldots, \mathtt{x}_k\}$ is a program variable, E is an arithmetic expression over program variables, $p \in [0,1]$ is a probability, and B is a predicate (called guard*) over program variables.*

The FPS semantics of pGCL will be defined in a forward denotational style, where the program variables $\mathtt{x}_1, \ldots, \mathtt{x}_k$ correspond to the formal indeterminates $X_1, \ldots, X_k$ of FPSs.

For handling assignments, `if`-conditionals and `while`-loops, we need some auxiliary functions on FPSs: For an arithmetic expression E over program variables, we denote by $\mathsf{eval}_\sigma(E)$ the evaluation of E in program state σ. For a predicate $B \subseteq \mathbb{N}^k$ and FPS F, we define the *restriction of F to B* by

$$\langle F \rangle_B \;:=\; \sum_{\sigma \in B} [\sigma]_F \cdot \mathbf{X}^\sigma \ ,$$

i.e. $\langle F \rangle_B$ is the FPS obtained from F by setting all coefficients $[\sigma]_F$ where $\sigma \notin B$ to 0. Using these prerequisites, our FPS transformer semantics is given as follows:

Definition 7 (FPS Semantics of pGCL). *The semantics $[\![P]\!] \colon \mathsf{FPS} \to \mathsf{FPS}$ of a loop-free pGCL program P is given according to the upper part of Table 1.*

The unfolding operator *$\Phi_{B,P}$ for the loop $\texttt{while}\,(B)\,\{P\}$ is defined by*

$$\Phi_{B,P}\colon \quad (\mathsf{FPS} \to \mathsf{FPS}) \to (\mathsf{FPS} \to \mathsf{FPS}), \quad \psi \;\mapsto\; \lambda F.\, \langle F \rangle_{\neg B} + \psi\Big([\![P]\!]\big(\langle F \rangle_B\big)\Big).$$

The partial order $(\mathsf{FPS}, \preceq)$ extends to a partial order $\big(\mathsf{FPS} \to \mathsf{FPS}, \sqsubseteq\big)$ on FPS transformers by a point-wise lifting of $\preceq$. The least element of this partial order is the transformer $\mathbf{0} = \lambda F.\, 0$ mapping any FPS F to the zero series. The semantics of $\texttt{while}\,(B)\,\{P\}$ is then given by the least fixed point (with respect to $\sqsubseteq$) of its unfolding operator, i.e. $[\![\texttt{while}\,(B)\,\{P\}]\!] \;=\; \mathsf{lfp}\ \Phi_{B,P}$.

Table 1. FPS transformer semantics of pGCL programs.

P	$[\![P]\!](F)$
`skip`	F
$\mathtt{x}_i \coloneqq E$	$\sum_{\sigma\in\mathbb{N}^k} \mu_\sigma X_1^{\sigma_1} \cdots X_i^{\mathsf{eval}_\sigma(E)} \cdots X_k^{\sigma_k}$
$\{P_1\}\,[p]\,\{P_2\}$	$p \cdot [\![P_1]\!](F) \;+\; (1-p)\cdot [\![P_2]\!](F)$
$\mathtt{if}\,(B)\,\{P_1\}\,\mathtt{else}\,\{P_2\}$	$[\![P_1]\!]\big(\langle F\rangle_B\big) \;+\; [\![P_2]\!]\big(\langle F\rangle_{\neg B}\big)$
$P_1 ; P_2$	$[\![P_2]\!]\big([\![P_1]\!](F)\big)$
`while`$(B)\{P\}$	$\big(\mathsf{lfp}\ \Phi_{B,P}\big)(F)$, for $\Phi_{B,P}(\psi) \;=\; \lambda F.\ \langle F\rangle_{\neg B} \;+\; \psi\Big([\![P]\!]\big(\langle F\rangle_B\big)\Big)$

Example 4. Consider the program $P = \{\mathtt{x} \coloneqq 0\}\,[1/2]\,\{\mathtt{x} \coloneqq 1\}; \mathtt{c} \coloneqq \mathtt{c}+1$ and the input PGF $G = 1$, which denotes a point mass on state $\sigma = \mathbf{0}$. Using the annotation style shown in the right margin, denoting that $[\![P']\!](G) = G'$, we calculate $[\![P]\!](G)$ as follows:

$$
\begin{aligned}
&/\!/\!/\ G\\
&P'\\
&/\!/\!/\ G'
\end{aligned}
$$

$$
\begin{aligned}
&/\!/\!/\ 1\\
&\{\mathtt{x} \coloneqq 0\}\,[1/2]\,\{\mathtt{x} \coloneqq 1\};\\
&/\!/\!/\ \tfrac{1}{2} + \tfrac{X}{2}\\
&\mathtt{c} \coloneqq \mathtt{c}+1\\
&/\!/\!/\ \tfrac{C}{2} + \tfrac{CX}{2}
\end{aligned}
$$

As for the semantics of $\mathtt{c} \coloneqq \mathtt{c}+1$, see Table 2. △

Before we study how our FPS transformers behave on PGFs in particular, we now first argue that our FPS semantics is well-defined. While evident for loop-free programs, we appeal to the Kleene Fixed Point Theorem for loops [24], which requires ω-continuous functions.

Theorem 1 (ω-continuity of pGCLSemantics). *The semantic functional $[\![\cdot]\!]$ is ω-continuous, i.e. for all programs $P \in$ pGCL and all increasing ω-chains $F_1 \preceq F_2 \preceq \ldots$ in* FPS*,*

$$[\![P]\!]\left(\sup_{n\in\mathbb{N}} F_n\right) \;=\; \sup_{n\in\mathbb{N}} [\![P]\!](F_n)\ .$$

Theorem 2 (Well-definedness of FPS Semantics). *The semantics functional* $[\![\cdot]\!]$ *is well-defined, i.e. the semantics of any loop* `while` $(B)\,\{P\}$ *exists uniquely and can be written as*

$$[\![\texttt{while}\,(B)\,\{P\}]\!] \;=\; \mathsf{lfp}\;\Phi_{B,P} \;=\; \sup_{n\in\mathbb{N}} \Phi^n_{B,P}(\mathbf{0})\;.$$

Table 2. Common assignments and their effects on the input PGF $F(X,Y)$.

P	$[\![P]\!](F)$
$\texttt{x} \coloneqq \texttt{x} + k$	$X^k \cdot F(X,Y)$
$\texttt{x} \coloneqq k \cdot \texttt{x}$	$F(X^k,Y)$
$\texttt{x} \coloneqq \texttt{x} + \texttt{y}$	$F(X,XY)$

4.2 Healthiness Conditions of FPS Transformers

In this section we show basic, yet important, properties which follow from [23]. For instance, for any input FPS F, the semantics of a program cannot yield as output an FPS with a mass larger than $|F|$, i.e. *programs cannot create mass.*

Theorem 3 (Mass Conservation). *For every* $P \in \mathtt{pGCL}$ *and* $F \in \mathsf{FPS}$*, we have* $\big|[\![P]\!](F)\big| \;\leq\; |F|$.

A program P is called *mass conserving* if $|[\![P]\!](F)| = |F|$ for all $F \in \mathsf{FPS}$. Mass conservation has important implications for FPS transformers acting on PGFs: given as input a PGF, the semantics of a program yields a PGF.

Corollary 1 (PGF Transformers). *For every* $P \in \mathtt{pGCL}$ *and* $G \in \mathsf{PGF}$*, we have* $[\![P]\!]\,(G) \in \mathsf{PGF}$.

Restricted to PGF, our semantics hence acts as a subdistribution transformer. Output masses may be smaller than input masses. The probability of nontermination of the programs is captured by the "missing" probability mass.

As observed in [23], semantics of probabilistic programs are fully defined by their effects on point masses, thus rendering probabilistic program semantics linear. In our setting, this generalizes to linearity of our FPS transformers.

Definition 8 (Linearity). *Let* $F, G \in \mathsf{FPS}$ *and* $r \in \mathbb{R}^\infty_{\geq 0}$ *be a scalar. The function* $\psi\colon \mathsf{FPS} \to \mathsf{FPS}$ *is called a* linear transformer *(or simply* linear*), if*

$$\psi(r \cdot F + G) \;=\; r \cdot \psi(F) \;+\; \psi(G)\;.$$

Theorem 4 (Linearity of `pGCL` Semantics). *For every program* P *and guard* B*, the functions* $\langle \cdot \rangle_B$ *and* $[\![P]\!]$ *are linear. Moreover, the unfolding operator* $\Phi_{B,P}$ *maps linear transformers onto linear transformers.*

As a final remark, we can unroll `while` loops:

Lemma 2 (Loop Unrolling). *For any FPS* F*,*

$$[\![\texttt{while}\,(B)\,\{P\}]\!]\,(F) \;=\; \langle F\rangle_{\neg B} \;+\; [\![\texttt{while}\,(B)\,\{P\}]\!]\,\big([\![P]\!]\big(\langle F\rangle_B\big)\big)\;.$$

4.3 Embedding into Kozen's Semantics Framework

Kozen [23] defines a generic way of giving distribution transformer semantics based on an abstract measurable space $(X^n, M^{(n)})$. Our FPS semantics instantiates his generic semantics. The state space we consider is $\mathbb{N}^k$, so that $(\mathbb{N}^k, \mathcal{P}(\mathbb{N}^k))$ is our measurable space.[3] A measure on that space is a countably-additive function $\mu\colon \mathcal{P}(\mathbb{N}^k) \to [0,\infty]$ with $\mu(\emptyset) = 0$. We denote the set of all measures on our space by $\mathcal{M}$. Although, we represent measures by FPSs, the two notions are in bijective correspondence $\tau\colon \mathsf{FPS} \to \mathcal{M}$, given by

$$\tau(F) \;=\; \lambda S. \sum_{\sigma \in S} [\sigma]_F \ .$$

This map preserves the linear structure and the order $\preceq$.

Kozen's syntax [23] is slightly different from pGCL. We compensate for this by a translation function $\mathfrak{T}$, which maps pGCL programs to Kozen's. The following theorem shows that our semantics agrees with Kozen's semantics.[4]

Theorem 5. *The* FPS *semantics of* pGCL *is an instance of Kozen's semantics, i.e. for all* pGCL *programs* P*, we have*

$$\tau \circ [\![P]\!] = \mathfrak{T}(P) \circ \tau \ .$$

Equivalently, the following diagram commutes:

$$\begin{array}{ccc} \mathsf{FPS} & \xrightarrow{\ \tau\ } & \mathcal{M} \\ {\scriptstyle [\![P]\!]}\downarrow & & \downarrow{\scriptstyle \mathfrak{T}(P)} \\ \mathsf{FPS} & \xrightarrow[\ \tau\ ]{} & \mathcal{M} \end{array}$$

For more details about the connection between FPSs and measures, as well as more information about the actual translation, see Appendix A.3.

5 Analysis of Probabilistic Programs

Our PGF semantics enables the representation of the effect of a pGCL program on a given PGF. As a next step, we investigate to what extent a program analysis can exploit such PGF representations. To that end, we consider the overapproximation with loop invariants (Sect. 5.1) and provide examples showing that checking whether an FPS transformer overapproximates a loop can be checked with computer algebra tools. In addition, we determine a subclass of pGCL programs whose effect on an arbitrary input state is ensured to be a rational PGF encoding a phase-type distribution (Sect. 5.2).

[3] We note that we want each point σ to be measurable, which enforces a *discrete* measurable space.

[4] Note that Kozen regards a program P itself as a function $P\colon \mathcal{M} \to \mathcal{M}$.

5.1 Invariant-Style Overapproximation of Loops

In this section, we seek to overapproximate loop semantics, i.e. for a given loop $W = \texttt{while}\,(B)\,\{P\}$, we want to find a (preferably simple) FPS transformer ψ, such that $[\![W]\!] \sqsubseteq \psi$, meaning that for any input G, we have $[\![W]\!](G) \preceq \psi(G)$ (cf. Definition 7). Notably, even if G is a PGF, we do not require $\psi(G)$ to be one. Instead, $\psi(G)$ can have a mass larger than one. This is fine, because it still overapproximates the actual semantics coefficient-wise. Such overapproximations immediately carry over to reading off expected values (cf. Sect. 3), for instance

$$\tfrac{\partial}{\partial X}[\![W]\!](G)(\mathbf{1}) \quad \leq \quad \tfrac{\partial}{\partial X}\psi(G)(\mathbf{1})\ .$$

We use invariant-style reasoning for verifying that a *given* ψ overapproximates the semantics of $[\![W]\!]$. For that, we introduce the notion of a *superinvariant* and employ Park's Lemma [30]—well-known in fixed point theory—to obtain a conceptually simple proof rule for verifying overapproximations of while loops.

Theorem 6 (Superinvariants and Loop Overapproximations). *Let $\Phi_{B,P}$ be the unfolding operator of $\texttt{while}(B)\{P\}$ (cf.* Def. 7*) and $\psi\colon \mathsf{FPS} \to \mathsf{FPS}$. Then*

$$\Phi_{B,P}(\psi) \sqsubseteq \psi \quad \text{implies} \quad [\![\texttt{while}\,(B)\,\{P\}]\!] \sqsubseteq \psi\ .$$

We call a ψ satisfying $\Phi_{B,P}(\psi) \sqsubseteq \psi$ a *superinvariant*. We are interested in linear superinvariants, as our semantics is also linear (cf. Theorem 4). Furthermore, linearity allows to define ψ solely in terms of its effect on monomials, which makes reasoning considerably simpler:

Corollary 2. *Given $f\colon \mathrm{Mon}(\mathbf{X}) \to \mathsf{FPS}$, let the* linear extension $\hat{f}$ of f *be*

$$\hat{f}\colon \quad \mathsf{FPS} \to \mathsf{FPS}, \quad F \mapsto \sum_{\sigma\in\mathbb{N}^k} [\sigma]_F\, f(\mathbf{X}^\sigma)\ .$$

Let $\Phi_{B,P}$ be the unfolding operator of $\texttt{while}\,(B)\,\{P\}$. Then

$$\forall\,\sigma \in \mathbb{N}^k\colon \quad \Phi_{B,P}(\hat{f})(\mathbf{X}^\sigma) \sqsubseteq \hat{f}(\mathbf{X}^\sigma) \qquad \text{implies} \qquad [\![\texttt{while}\,(B)\,\{P\}]\!] \sqsubseteq \hat{f}\ .$$

We call an f satisfying the premise of the above corollary a *superinvariantlet*. Notice that superinvariantlets and their extensions agree on monomials, i.e. $f(\mathbf{X}^\sigma) = \hat{f}(\mathbf{X}^\sigma)$. Let us examine a few examples for superinvariantlet-reasoning.

Example 5 (Verifying Precise Semantics). In Program 1.1, in each iteration, a fair coin flip determines the value of `x`. Subsequently, `c` is incremented by 1. Consider the following superinvariantlet:

$$f(X^iC^j) \;=\; C^j \cdot \begin{cases} \frac{C}{2-C}, & \text{if } i = 1; \\ X^i, & \text{if } i \neq 1. \end{cases}$$

```
while (x = 1){
  {x ≔ 0} [1/2] {x ≔ 1};
  c ≔ c + 1
}
```

Program 1.1. Geometric distribution generator.

To verify that f is indeed a superinvariantlet, we have to show that

$$\begin{aligned}\Phi_{B,P}(\hat{f})(X^iC^j) &= \langle X^iC^j\rangle_{x\neq 1} + \hat{f}\left([\![P]\!]\left(\langle X^iC^j\rangle_{x=1}\right)\right)\\ &\overset{!}{\sqsubseteq} \hat{f}\left(X^iC^j\right) .\end{aligned}$$

For $i \neq 1$, we get

$$\begin{aligned}\Phi_{B,P}(\hat{f})(X^iC^j) &= \langle X^iC^j\rangle_{x\neq 1} + \hat{f}([\![P]\!](0))\\ &= X^iC^j = f(X^iC^j) = \hat{f}(X^iC^j) .\end{aligned}$$

For $i = 1$, we get

$$\begin{aligned}\Phi_{B,P}(\hat{f})(X^1C^j) &= \hat{f}\left(\tfrac{1}{2}X^0C^{j+1} + \tfrac{1}{2}X^1C^{j+1}\right)\\ &= \tfrac{1}{2}f\left(X^0C^{j+1}\right) + \tfrac{1}{2}f\left(X^1C^{j+1}\right) && \text{(by linearity of } \hat{f}\text{)}\\ &= \tfrac{C^{j+1}}{2-C} = f\left(X^1C^j\right) = \hat{f}\left(X^1C^j\right) . && \text{(by definition of } f\text{)}\end{aligned}$$

Hence, Corollary 2 yields $[\![W]\!](X) \sqsubseteq f(X) = \frac{C}{2-C}$.

For this example, we can state even more. As the program is almost surely terminating, and $\left|f(X^iC^j)\right| = 1$ for all $(i,j) \in \mathbb{N}^2$, we conclude that $\hat{f}$ is exactly the semantics of W, i.e. $\hat{f} = [\![W]\!]$. △

```
while (x > 0){
  {x ≔ x + 1} [1/2] {x ≔ x - 1};
  c ≔ c + 1
}
```

Program 1.2. Left-bounded 1-dimensional random walk.

Example 6 (Verifying Proper Overapproximations). Program 1.2 models a one dimensional, left-bounded random walk. Given an input $(i,j) \in \mathbb{N}^2$, this program can only terminate in an even (if i is even) or odd (if i is odd) number of steps. This insight can be encoded into the following superinvariantlet:

$$\begin{aligned}f(X^0C^j) &= C^j \quad \text{and}\\ f(X^{i+1}C^j) &= C^j \cdot \begin{cases}\frac{C}{1-C^2}, & \text{if } i \text{ is odd;}\\ \frac{1}{1-C^2}, & \text{if } i \text{ is even.}\end{cases}\end{aligned}$$

It is straightforward to verify that f is a *proper* superinvariantlet (proper because $\frac{C}{1-C^2} = C + C^3 + C^5 + \ldots$ is *not* a PGF) and hence f *properly* overapproximates the loop semantics. Another superinvariantlet for Program 1.2 is given by

$$h(X^i C^j) \;=\; C^j \cdot \begin{cases} \left(\frac{1-\sqrt{1-C^2}}{C}\right)^i, & \text{if } i \geq 1; \\ 1, & \text{if } i = 0. \end{cases}$$

Given that the program terminates almost-surely [16] and that h is a superinvariantlet yielding only PGFs, it follows that the extension of h is *exactly* the semantics of Program 1.2. An alternative derivation of this formula for the case $h(X)$ can be found, e.g., in [17].

For both f and h, we were able to prove that they are indeed superinvariantlets *semi-automatically*, using the computer algebra library SymPy [27]. The code is included in Appendix B (Program 1.5). △

```
while (x > 0){
        {x := x - 1} [1/x] {x := x + 1}
}
```

Program 1.3. A non-almost-surely terminating loop.

Example 7 (Proving Non-almost-sure Termination). In Program 1.3, the branching probability of the choice statement depends on the value of a program variable. This notation is just syntactic sugar, as this behavior can be mimicked by loop constructs together with coin flips [3, pp. 115f].

To prove that Program 1.3 does *not* terminate almost-surely, we consider the following superinvariantlet:

$$f(X^i) \;=\; 1 - \frac{1}{e} \cdot \sum_{n=0}^{i-2} \frac{1}{n!}\,, \qquad \text{where } e = 2.71828\ldots \text{ is Euler's number.}$$

Again, the superinvariantlet property was *verified semi-automatically*, by this we mean that we have constructed functions f and Φ by hand and Mathematica [18] confirmed that $\Phi(f) - f = 0$. Now, consider for instance $f(X^3) = 1 - \frac{1}{e} \cdot \left(\frac{1}{0!} + \frac{1}{1!}\right) = 1 - \frac{2}{e} < 1$. This proves, that the program terminates on X^3 with a probability strictly smaller than 1, witnessing that the program is not almost surely terminating. Note that in general this technique cannot be used for proving almost-sure termination. △

5.2 Rational PGFs

In several of the examples from the previous sections, we considered PGFs which were *rational functions*, that is, fractions of two polynomials. Since those are a

particularly simple class of PGFs, it is natural to ask which programs have rational semantics. In this section, we present a semantic characterization of a class of `while`-loops whose output distribution is a (multivariate) *discrete phase-type* distribution [28,29]. This implies that the resulting PGF of such programs is an effectively computable rational function for any given input state. Let us illustrate this by an example.

```
while (x < 1 and t < 2){
        if (t = 0){
                {x := 1} [a] {t := 1}; c := c + 1
        } else {
                {x := 1} [b] {t := 0}; d := d + 1
        }
}
```

Program 1.4. Dueling cowboys.

Example 8 (Dueling Cowboys). Program 1.4 models two dueling cowboys [25]. The hit chance of the first cowboy is a and the hit chance of the second cowboy is b, where $a, b \in [0,1]$.[5] The cowboys shoot at each other in turns, as indicated by the variable `t`, until one of them gets hit (`x` is set to `1`). The variable `c` counts the number of shots of the first cowboy and `d` those of the second cowboy.

We observe that Program 1.4 is somewhat independent of the value of `c`. More specifically, placing the additional statement `c := c + 1` either immediately before or after the loop yields two equivalent programs. In our notation, this is expressed as $[\![W]\!](C \cdot H) = C \cdot [\![W]\!](H)$ for all PGFs H. By symmetry, the same applies to variable `d`. Unfolding the loop once on input 1, yields

$$[\![W]\!](1) \;=\; (1-a)C \cdot [\![W]\!](T) + aCX \;.$$

A similar equation for $[\![W]\!](T)$ involving $[\![W]\!](1)$ on its right-hand side holds. This way we obtain a system of two linear equations, although the program itself is infinite-state. The linear equation system has a unique solution $[\![W]\!](1)$ in the field of rational functions over the variables C, D, T, and X which is the PGF

$$G \;\coloneqq\; \frac{aCX + (1-a)bCDTX}{1-(1-b)(1-a)CD} \;.$$

From G we can easily read off the following: The probability that the first cowboy wins (`x = 1` and `t = 0`) equals $\frac{a}{1-(1-a)(1-b)}$, and the expected total number of shots of the first cowboy is $\frac{\partial}{\partial C}G(1) = \frac{1}{a+b-ab}$. Notice that this quantity equals ∞ if a and b are both zero, i.e. if both cowboys have zero hit chance.

If we write $G_{\mathbf{V}}$ for the PGF obtained by substituting all but the variables in $\mathbf{V}$ with 1, then we moreover see that $G_C \cdot G_D \neq G_{C,D}$. This means that C and D (as random variables) are stochastically dependent. △

[5] These are *not* program variables.

The distribution encoded in the PGF $[\![W]\!](1)$ is a discrete phase-type distribution. Such distributions are defined as follows: A *Markov reward chain* is a Markov chain where each state is augmented with a reward vector in $\mathbb{N}^k$. By definition, a (discrete) distribution on $\mathbb{N}^k$ is of phase-type iff it is the distribution of the total accumulated reward vector until absorption in a Markov reward chain with a single absorbing state and a finite number of transient states. In fact, Program 1.4 can be described as a Markov reward chain with two states (X^0T^0 and X^0T^1) and 2-dimensional reward vectors corresponding to the "counters" $(\mathtt{c}, \mathtt{d})$: the reward in state X^0T^0 is $(1, 0)$ and $(0, 1)$ in the other state.

Each pGCL program describes a Markov reward chain [13]. It is not clear which (non-trivial) syntactical restrictions to impose to guarantee for such chains to be finite. In the remainder of this section, we give a characterization of `while`-loops that are equivalent to finite Markov reward chains. The idea of our criterion is that each variable has to fall into one of the following two categories:

Definition 9 (Homogeneous and Bounded Variables). *Let $P \in$ pGCL be a program, B be a guard and $\mathtt{x}_i$ be a program variable. Then:*

- $\mathtt{x}_i$ *is called* homogeneous *for P if $[\![P]\!](X_i \cdot G) = X_i \cdot [\![P]\!](G)$ for all $G \in \mathsf{PGF}$.*
- $\mathtt{x}_i$ *is called* bounded *by B if the set $\{\sigma_i \mid \sigma \in B\}$ is finite.*

Intuitively, homogeneity of $\mathtt{x}_i$ means that it does not matter whether one increments the variable before or after the execution of P. Thus, a homogeneous variable *behaves like an increment-only counter* even if this may not be explicit in the syntax. In Example 8, the variables `c` and `d` in Program 1.4 are homogeneous (for both the loop-body and the loop itself). Moreover, `x` and `t` are clearly bounded by the loop guard. We can now state our characterization.

Definition 10 (HB Loops). *A loop* `while` (B) $\{P\}$ *is called* homogeneous-bounded (HB) *if for all program states $\sigma \in B$, the PGF $[\![P]\!](\mathbf{X}^\sigma)$ is a polynomial and for all program variables* `x` *it* either *holds that*

- `x` *is homogeneous for P and the guard B is independent of* `x`*, or that*
- `x` *is bounded by the guard B.*

In an HB loop, all the possible valuations of the bounded variables satisfying B span the *finite* transient state space of a Markov reward chain in which the dimension of the reward vectors equals the number of homogeneous variables. The additional condition that $[\![P]\!](\mathbf{X}^\sigma)$ is a polynomial ensures that there is only a finite amount of terminal (absorbing) states. Thus, we have the following:

Proposition 1. *Let W be a while-loop. Then $[\![W]\!](\mathbf{X}^\sigma)$ is the (rational) PGF of a multivariate discrete phase-type distribution if and only if W is equivalent to an HB loop that almost-surely terminates on input σ.*

To conclude, we remark that there are various simple *syntactic* conditions for HB loops: For example, if P is loop-free, then $[\![P]\!](\mathbf{X}^\sigma)$ is always a polynomial. Similarly, if `x` only appears in assignments of the form `x := x +` k, $k \geq 0$,

then x is homogeneous. Such updates of variables are e.g. essential in *constant probability programs* [11]. The crucial point is that such conditions are only sufficient but not necessary. Our *semantic* conditions thus capture the essence of phase-type distribution semantics more adequately while still being reasonably simple (albeit—being non-trivial semantic properties—undecidable in general).

6 Conclusion

We have presented a denotational distribution transformer semantics for probabilistic while-programs where the denotations are generating functions (GFs). The main benefit of using GFs lies in representing the entire probability distribution for a given input. Moreover, we have provided a simple invariant-style technique to prove that a given GF overapproximates the program's semantics and identified a class of (possibly infinite-state) programs whose semantics is a rational GF encoding a discrete phase-type distribution. Directions for future work include the (semi-)automated synthesis of invariants and the development of notions on how precise overapproximations by invariants actually are. On that end, a rule for verifying *under*approximations (e.g. à la [14], which provides inductive rules for underapproximating expected values) would be a major step in that direction.

Another direction for future work is to support $\mathbb{Z}$-valued program variables. For expected values, work on verifying signed random variables exists [21]—for PGFs, the situation is less clear. An obvious choice would be to employ formal *Laurent series*, but those only allow for *finitely many* negative indices, thus eluding distributions with both infinite positive and infinite negative support.

Acknowledgements. The authors thank the reviewers for their constructive and helpful comments and Marcel Hark for fruitful discussions.

References

1. Barthe, G., Grégoire, B., Hsu, J., Strub, P.: Coupling proofs are probabilistic product programs. In: POPL, pp. 161–174. ACM (2017)
2. Bartocci, E., Kovács, L., Stankovič, M.: Automatic generation of moment-based invariants for prob-solvable loops. In: Chen, Y.-F., Cheng, C.-H., Esparza, J. (eds.) ATVA 2019. LNCS, vol. 11781, pp. 255–276. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-31784-3_15
3. Batz, K., Kaminski, B.L., Katoen, J., Matheja, C., Noll, T.: Quantitative separation logic: a logic for reasoning about probabilistic pointer programs. In: PACMPL 3 (POPL), pp. 34:1–34:29 (2019)
4. Boreale, M.: Analysis of probabilistic systems via generating functions and Padé approximation. In: Halldórsson, M.M., Iwama, K., Kobayashi, N., Speckmann, B. (eds.) ICALP 2015. LNCS, vol. 9135, pp. 82–94. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-47666-6_7

5. Bouissou, O., Goubault, E., Putot, S., Chakarov, A., Sankaranarayanan, S.: Uncertainty propagation using probabilistic affine forms and concentration of measure inequalities. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 225–243. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_13
6. 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
7. Chatterjee, K., Fu, H., Novotný, P., Hasheminezhad, R.: Algorithmic analysis of qualitative and quantitative termination problems for affine probabilistic programs. ACM Trans. Program. Lang. Syst. **40**(2), 7:1–7:45 (2018)
8. Cho, K., Jacobs, B.: The EfProb library for probabilistic calculations. In: CALCO. LIPIcs, vol. 72, pp. 25:1–25:8. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2017)
9. Cousot, P., Monerau, M.: Probabilistic abstract interpretation. In: Seidl, H. (ed.) ESOP 2012. LNCS, vol. 7211, pp. 169–193. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28869-2_9
10. Di Pierro, A., Wiklicky, H.: Semantics of probabilistic programs: a weak limit approach. In: Shan, C. (ed.) APLAS 2013. LNCS, vol. 8301, pp. 241–256. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-03542-0_18
11. Giesl, J., Giesl, P., Hark, M.: Computing expected runtimes for constant probability programs. In: Fontaine, P. (ed.) CADE 2019. LNCS (LNAI), vol. 11716, pp. 269–286. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29436-6_16
12. Graham, R., Knuth, D., Patashnik, O.: Concrete Mathematics: A Foundation for Computer Science. Addison-Wesley, Boston (1994)
13. Gretz, F., Katoen, J., McIver, A.: Operational versus weakest pre-expectation semantics for the probabilistic guarded command language. Perform. Eval. **73**, 110–132 (2014)
14. 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)
15. den Hartog, J., de Vink, E.P.: Verifying probabilistic programs using a Hoare like logic. Int. J. Found. Comput. Sci. **13**(3), 315–340 (2002)
16. Hurd, J.: A formal approach to probabilistic termination. In: Carreño, V.A., Muñoz, C.A., Tahar, S. (eds.) TPHOLs 2002. LNCS, vol. 2410, pp. 230–245. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45685-6_16
17. Icard, T.: Calibrating generative models: the probabilistic Chomsky-Schützenberger hierarchy. J. Math. Psychol. **95**, 102308 (2020). https://www.sciencedirect.com/journal/journal-of-mathematical-psychology/vol/95/suppl/C
18. Inc., W.R.: Mathematica, Version 12.0, champaign, IL (2019). https://www.wolfram.com/mathematica
19. Johnson, N., Kotz, S., Kemp, A.: Univariate Discrete Distributions. Wiley, Hoboken (1993)
20. Kaminski, B.L.: Advanced weakest precondition calculi for probabilistic programs. Ph.D. thesis, RWTH Aachen University, Germany (2019)
21. Kaminski, B.L., Katoen, J.: A weakest pre-expectation semantics for mixed-sign expectations. In: ACM/IEEE Symposium on Logic in Computer Science. LICS, pp. 1–12. IEEE Computer Society (2017)
22. 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

23. Kozen, D.: Semantics of probabilistic programs. In: FOCS, pp. 101–114. IEEE Computer Society (1979)
24. Lassez, J.L., Nguyen, V.L., Sonenberg, L.: Fixed point theorems and semantics: a folk tale. Inf. Process. Lett. **14**(3), 112–116 (1982)
25. McIver, A., Morgan, C.: Abstraction, refinement and proof for probabilistic systems. Monogr. Comput. Sci. Springer (2005). https://doi.org/10.1007/b138392
26. McIver, A., Morgan, C., Kaminski, B.L., Katoen, J.: A new proof rule for almost-sure termination. In: PACMPL **2**(POPL), pp. 33:1–33:28 (2018)
27. Meurer, A., et al.: SymPy: symbolic computing in python. PeerJ Comput. Sci. **3**, e103 (2017). https://doi.org/10.7717/peerj-cs.103
28. Navarro, A.C.: Order statistics and multivariate discrete phase-type distributions. Ph.D. thesis, DTU Lyngby (2018)
29. Neuts, M.F.: Matrix-geometric solutions to stochastic models. In: Steckhan, H., Bühler, W., Jäger, K.E., Schneeweiß, C., Schwarze, J. (eds.) DGOR, pp. 425–425. Springer, Heidelberg (1984). https://doi.org/10.1007/978-3-642-69546-9_91
30. Park, D.: Fixpoint induction and proofs of program properties. Mach. Intell. **5**, 59–78 (1969)
31. Pólya, G.: Mathematics and Plausible Reasoning: Induction and Analogy in Mathematics. Princeton University Press, Princeton (1954)
32. Wiklicky, H.: On dynamical probabilities, or: how to learn to shoot straight. In: Lluch Lafuente, A., Proença, J. (eds.) COORDINATION 2016. LNCS, vol. 9686, pp. 262–277. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-39519-7_16