# Exact Bayesian Inference for Loopy Probabilistic Programs

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

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

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

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

**ACM Reference Format:**
Lutz Klinkenberg, Christian Blumenthal, Mingshuai Chen, Darion Haase, and Joost-Pieter Katoen. 2024. Exact Bayesian Inference for Loopy Probabilistic Programs. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 1 (January 2024), 51 pages.

## 1 INTRODUCTION

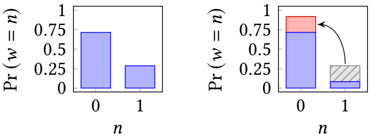
Probabilistic programming is used to describe stochastic models in the form of executable computer programs. It enables fast and natural ways of designing statistical models without ever resorting to random variables in the mathematical sense. The so-obtained probabilistic programs [Barthe et al. 2020; Gordon et al. 2014; Holtzen et al. 2020; Kozen 1981; van de Meent et al. 2018] are typically normal-looking programs describing posterior probability distributions. They intrinsically code up randomized algorithms [Mitzenmacher and Upfal 2005] and are at the heart of approximate computing [Carbin et al. 2016] as well as probabilistic machine learning [van de Meent et al. 2018, Chapter 8]. One prominent example is Scenic [Fremont et al. 2022] – a domain-specific probabilistic programming language to describe and generate scenarios for, e.g., robotic systems, that can be used to train convolutional neural networks; Scenic features the ability to declaratively impose (hard and soft) constraints over the generated models by means of *conditioning* via posterior observations. Moreover, a large volume of literature has been devoted to combining the strength of probabilistic and differentiable programming in a mutually beneficial manner; see [van de Meent et al. 2018, Chapter 8] for recent advancements in deep probabilistic programming.

Reasoning about probabilistic programs amounts to addressing various *quantities* like assertion-violation probabilities [Wang et al. 2021b], preexpectations [Batz et al. 2021; Feng et al. 2023; Hark et al. 2020], moments [Moosbrugger et al. 2022; Wang et al. 2021a], expected runtimes [Kaminski

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

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

Prog. 1. The telephone operator.



(a) initial belief (b) updated belief

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

et al. 2018], and concentrations [Chakarov and Sankaranarayanan 2013; Chatterjee et al. 2016]. Probabilistic inference is one of the most important tasks in quantitative reasoning which aims to derive a program's posterior distribution. In contrast to sampling-based approximate inference, inferring the *exact* distribution has several benefits [Gehr et al. 2020], e.g., no loss of precision, natural support for symbolic parameters, and efficiency on models with certain structures.

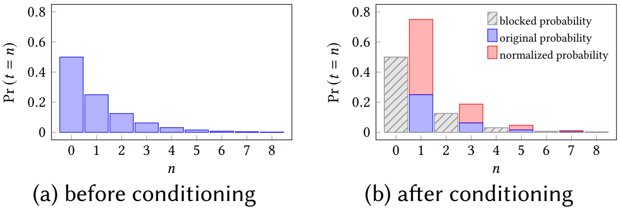
Exact probabilistic inference, however, is a notoriously difficult task [Ackerman et al. 2019; Cooper 1990; Kaminski et al. 2019; Olmedo et al. 2018; Roth 1996]; even for Bayesian networks, it is already PP-complete [Kwisthout 2009; Littman et al. 1998]. The challenges mainly arise from three program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

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

*Conditioning in loop-free programs.* Consider the loop-free program Prog. 1 producing an *infinite-support* distribution. It describes a telephone operator who is unaware of whether today is a weekday or weekend. The operator's initial belief is that with probability $5/7$ it is a weekday ($w = 0$) and thus with probability $2/7$ weekend ($w = 1$); see Fig. 1a. Usually, on weekdays there are 6 incoming calls per hour on average; on weekends this rate decreases to 2 calls – both rates are subject to a Poisson distribution. The operator observes 5 calls in the last hour, and the inference task is to compute the distribution in which the initial belief is updated based on the posterior observation. Our approach can automatically infer the updated belief (see Fig. 1b) with $\Pr(w = 0) = \frac{1215}{1215+2\cdot e^4} \approx 0.9175$. (Detailed calculations of the PGF semantics for Prog. 1 are given in Example 6 on page 10.)

```
$h := 1$ ⨟
while ( $h = 1$ ) {
   { $t := t + 1$ } [ $1/2$ ] { $h := 0$ }
} ⨟
observe ( $t \equiv 1 \pmod 2$ )
```

Prog. 2. The odd geometric distribution.



(a) before conditioning (b) after conditioning

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

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

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

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

*Conditioning inside i.i.d. loops.* As argued by Olmedo et al. [2018] and Bichsel et al. [2018], having loops and conditioning intertwined incurs semantic intricacies: Consider Prog. 3 – a variant of Prog. 2 where instead we observe $h = 1$ *inside* the while-loop. Although Prog. 3 features an *i.i.d. loop*, i.e., the set of states reached upon the end of different loop iterations are *independent and identically distributed*, assigning a meaningful semantics to this program is delicate, since we condition to a *zero-probability event*, i.e., the infinite program run that constantly visits the left branch of the probabilistic choice. Intuitively, the observe statement prevents the while-loop from ever terminating since we always observe that we have taken the left branch, and therefore never set the termination flag $h = 0$. As a consequence, all runs which eventually would have been terminated are no longer valid as they all violate the observation criterion, whereas the single run that does satisfy the criterion in turn is never able to exit the loop (cf. Section 3.2). In previous work on using PGFs [Chen et al. 2022a; Klinkenberg et al. 2020], observe violations are not considered and the semantics of non-termination is represented as subprobability distributions where the "missing" probability mass captures the probability of divergence. Zaiser et al. [2023] circumvent such semantic intricacies by syntactically imposing certainly terminating programs (due to the

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

Prog. 3. observe inside loop.

absence of loops and recursion). In our approach, we distinguish non-termination behaviors from `observe` violations, which allows us to show that the `while`-loop in Prog. 3 is in fact equivalent to `if` $(h = 1)$ {`observe` (`false`)} `else` {`skip`} $\equiv$ `observe` $(h \neq 1)$.

Alternative approaches detour the abovementioned semantic intricacies by transforming the conditioned loop into an equivalent loop without observations inside. These approaches include (1) *hoisting* [Olmedo et al. 2018] that removes observations completely from conditioned probabilistic programs, which however relies on highly intractable fixed point computations to hoist `observe` statements inside loops; (2) the *pre-image transformation* [Nori et al. 2014] that propagates observations backward through the program, which however cannot hoist the `observe` statement through probabilistic choices, as in Prog. 3; (3) the *ad hoc solution* that simply pulls the `observe` statement outside the loop, which however works only for special i.i.d. loops like Prog. 3: The `observe` statement in Prog. 3 can be equivalently moved downward to the outside of the loop, but such transformation does not generalize to non-i.i.d. loops (which may have data flow across different loop iterations) as exemplified below.

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

```
$n := 0$ ⨾
$h_1 := 1$ ⨾ $h_2 := 1$ ⨾ $h_1' := 1$ ⨾ $h_2' := 1$ ⨾
while ( $\neg (h_1 = 0 \wedge h_2 = 0)$ ) {
    $h_1 :=$ bernoulli ($1/2$) ⨾
    $h_2 :=$ bernoulli ($1/2$) ⨾
    observe ( $h_1 = h_1' \vee h_2 = h_2'$ ) ⨾
    $h_1' := h_1$ ⨾
    $h_2' := h_2$ ⨾
    $n := n + 1$
}
```

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

**Contributions.** The main results of this paper are:

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

while-loop coincides with inference for a straight-line program. Technically this is based on proving program equivalence.

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

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

## 2 PRELIMINARIES ON GENERATING FUNCTIONS

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## 3 GF SEMANTICS WITH CONDITIONING

Given a fixed input, the semantics of a probabilistic program is captured by its (posterior) probability distribution over the final (terminating) program states. In [Klinkenberg et al. 2020], the domain of discrete distributions is represented in terms of PGFs – elements from $\mathbb{R}[[\mathbf{X}]]$ – and a (conditioning-free) program is interpreted denotationally as a *distribution transformer* à la Kozen [Kozen 1981]. This representation comes with several benefits: (1) it naturally encodes common, *infinite-support* distributions like the geometric or Poisson distribution in compact, *closed-form* representations; (2) it allows for compositional reasoning and, in particular, in contrast to representations in terms of density or mass functions, the effective computation of (high-order) moments; (3) tail bounds, concentration bounds, and other properties of interest can be extracted with relative ease from a PGF; and (4) expressions containing parameters are naturally supported.

In order to carry these benefits forward to discrete, loopy probabilistic programs *with conditioning*, we extend the PGF semantics of Klinkenberg et al. [2020] to cope with posterior observations. To define such a semantic model, we fix $k$ $\mathbb{N}$-valued program variables $x_1, x_2, \ldots, x_k$. The set of program states is $\mathbb{N}^k$, where for each $\sigma = (\sigma_1, \ldots, \sigma_k) \in \mathbb{N}^k$, $\sigma_i$ indicates the value of $x_i$.

We consider the pGCL programming language [McIver and Morgan 2005] with the extended ability to specify posterior observations via the observe statements [Gordon et al. 2014; Nori et al. 2014; Olmedo et al. 2018]:

**Definition 2** (cpGCL)**.** *A program $P$ in the* conditional probabilistic guarded command language (cpGCL) *adheres to the grammar*

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

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

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

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

[2] We do not give an explicit syntax for $E$ and $B$ as it is irrelevant at this point. When dealing with *automation*, we present a concrete syntax, cf. Table 3 on page 13. Moreover, we write $\sigma \models B$ and $\sigma \in B$ interchangeably, meaning that the program state $\sigma$ satisfies the predicate $B$.

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

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

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

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

We emphasize that $|F| = F(\mathbf{1})$ does not take the `observe`-violation probability $[\lightning]_F$ into account. Indeterminates that are not related to program variables stay also unchanged in $F(\mathbf{1})$. Addition and scalar multiplication in eFPS are to be understood coefficient-wise, that is, for any $F, G \in$ eFPS,

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

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

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

We endow eFPSs and ePGFs with the following ordering relations.

**Definition 4 (Orders over** eFPS**).** *For all* $F, G \in$ eFPS*, let*

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

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

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

In fact, (eFPS, $\preceq$) and (eFPS $\to$ eFPS, $\sqsubseteq$) are complete lattices (cf. Appendix B). To evaluate Boolean guards, we use the so-called *filtering* function for eFPSs. The filtering of $F \in$ eFPS by predicate $B$ is

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

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

### 3.1 Unconditioned Semantics for `cpGCL`

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

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

[3] The coefficients $[\cdot]_F$ range over $\mathbb{R}_{\geq 0}^{\infty}$ to enforce a complete lattice structure over eFPS; see details in Appendices A and B.

Table 2. The *unconditioned* semantics for cpGCL programs.

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

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

Note that the observation-violation term $[\lightning]_G X_\lightning$ is just passed through all instructions but `observe` $(B)$. This renders the semantics as a conservative extension to [Klinkenberg et al. 2020], as for `observe`-free programs on initial distributions without $[\lightning]_G X_\lightning$, both semantics coincide.

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

### 3.2 Conditioned Semantics for cpGCL

The unconditioned semantics serves as an intermediate result to achieve our *conditioned* semantics, which further addresses *normalization* of distributions.

**Definition 5 (Normalization).** *The* normalization operator *norm is a partial function defined as*[4]

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

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

$$norm\left(\llbracket P \rrbracket\right) \ \triangleq \ \lambda G.\ norm\left(\llbracket P \rrbracket(G)\right) \ = \ \lambda G.\ \frac{\langle \llbracket P \rrbracket(G)\rangle_{\mathtt{true}}}{1 - [\lightning]_{\llbracket P \rrbracket(G)}}\,, \qquad \text{provided } [\lightning]_{\llbracket P \rrbracket_G} < 1.$$

[4] *norm* in fact maps an eFPS to an FPS, i.e., $[\lightning]_F X_\lightning$ is pruned away by normalization.

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

**Example 6 (Telephone Operator).** Reconsider Prog. 1, the loop-free program generating an infinite-support distribution. It describes a telephone operator who lacks knowledge about whether it is a weekday or weekend today. The operator's initial assumption is that there is a 5/7 probability of it being a weekday ($w = 0$) and a 2/7 probability of it being a weekend ($w = 1$). Typically, on weekdays, there are an average of 6 incoming calls per hour, while on weekends, this rate decreases to 2 calls. Both rates are governed by a Poisson distribution. The operator has observed 5 calls in the past hour, and the objective is to determine the updated distribution of the initial belief based on this posterior observation. We employ the forward annotation style as per [Kaminski 2019; Klinkenberg et al. 2020] and start the computation with prior distribution (eFPS) 1, which initializes every program variable to 0 with probability 1. By computing the transformations forward in sequence for each program instruction (see Prog. 5), we obtain the *unconditioned* semantics:

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

Normalizing this yields

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

```
/// $1$            $(= 1 \cdot W^0 C^0 + 0 \cdot X_{\lightning})$
$\{ w := 0 \}$ [ 5/7 ] $\{ w := 1 \}$ ⨾
/// $\frac{5}{7}W^0 + \frac{2}{7}W^1$
if ( $w = 0$ ) {
    /// $\frac{5}{7}$
    $c :=$ poisson (6)
    /// $\frac{5}{7}e^{-6(1-C)}$
} else {
    /// $\frac{2}{7}W$
    $c :=$ poisson (2)
    /// $\frac{2}{7}e^{-2(1-C)}W$
} ⨾
/// $\frac{5}{7}e^{-6(1-C)} + \frac{2}{7}e^{-2(1-C)}W$
observe ( $c = 5$ )
/// $\frac{(4860+8e^4W)}{105e^6}C^5 + (1 - \frac{4860+8e^4}{105e^6})X_{\lightning}$
```

Prog. 5. Semantics for the tel. operator.

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

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

[6] Programs that terminate with probability 1 on all inputs; see Section 4.1.

**Theorem 7 (Equivalence of Semantics).** *For every* cpGCL *program $P$, let $\mathcal{R}_\sigma[\![P]\!]$ be the Markov chain of $P$ starting in state $\sigma \in \mathbb{N}^k$. Then, for any $\sigma' \in \mathbb{N}^k$,*

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

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

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

## 4 EXACT BAYESIAN INFERENCE WITH LOOPS

Loops significantly complicate inferring posterior distributions of probabilistic programs. Computing the exact least fixed point of the characteristic function $\Phi_{B,P}$ is in general highly intractable, and thus other techniques like *invariant*-based reasoning are used. This section presents a program equivalence approach for a restricted set of *unconditioned* cpGCL programs, called cReDiP (cf. Table 3), to reason about loop invariants. We also use this technique to enable invariant synthesis by means of solving equation systems obtaining parameter values satisfying the invariant properties.

### 4.1 Invariant-Based Reasoning with Conditioning

Given a while-loop while ( $B$ ) { $P$ }, we call an eFPS transformer $I$: eFPS $\longrightarrow$ eFPS an *invariant* if $\Phi_{B,P}(I) = I$, i.e., it remains unchanged when pushed through one loop iteration. Moreover, to develop invariant-based reasoning techniques, we introduce the notion of lossless eFPS transformers:

**Definition 8 (Lossless eFPS Transformers).** *An eFPS transformer $H$:* eFPS $\longrightarrow$ eFPS *is* lossless *for $F \in$* eFPS *if*

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

*$H$ is* universally lossless *if it is lossless for all eFPS in* eFPS*.*

Since the semantics of a program $P$ is an eFPS transformer, $[\![P]\!]$ being (universally) lossless coincides with $P$ being (universally) almost-surely terminating, abbreviated as (U)AST [Bournez and Garnier 2005; Saheb-Djahromi 1978]. Given $L =$ while ( $B$ ) { $P$ }, we can approximate its least fixed point lfp $\Phi_{B,P}$ leveraging domain theory, in particular, Park's lemma, namely, $\Phi_{B,P}(I) \sqsubseteq I$ implies $[\![L]\!] \sqsubseteq I$ [Park 1969]. It enables reasoning about while-loops in terms of over-approximations and – in case a program is UAST– also about program equivalence.

**Theorem 9 (Loop Invariants).** *Given $L =$* while ( $B$ ) { $P$ } *and a universally lossless eFPS transformer $I$:* eFPS $\longrightarrow$ eFPS*. We have*

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

$$[\![L]\!] \;=\; I \quad \textit{and} \quad norm([\![L]\!](F)) \;=\; norm(I(F))\,.$$

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

$$F \preceq G \quad\implies\quad [\lightning]_F \le [\lightning]_G \quad\text{and}\quad \sum_{\sigma\in\mathbb{N}^k}[\sigma]_F\mathbf{X}^\sigma \preceq \sum_{\sigma\in\mathbb{N}^k}[\sigma]_G\mathbf{X}^\sigma$$

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

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

For (2), since $I$ is an invariant (i.e., a fixed point), $I$ must be at least lfp $\Phi_{B,P} = \llbracket \texttt{while}\,(\,B\,)\,\{\,P\,\}\rrbracket$. Moreover, because $\texttt{while}\,(\,B\,)\,\{\,P\,\}$ is UAST, it follows that

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

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

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

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

Effectively, reasoning about loops is reduced to *two challenges*: (1) finding an invariant candidate $I\colon \mathsf{eFPS} \longrightarrow \mathsf{eFPS}$ and (2) verifying that $I$ indeed is a valid invariant, i.e., deciding whether $\Phi_{B,P}(I) = I$. Since the semantics of a program $P$ is also of type $\mathsf{eFPS} \longrightarrow \mathsf{eFPS}$, we can describe such candidate by means of a program. In the remainder of this section we focus on *verifying* given invariant candidates (in the form of programs) while deferring finding invariants to Section 5.

## 4.2 Verification of Loop Invariants

Checking whether loop-free program $I$ is an invariant of $\texttt{while}\,(\,B\,)\,\{\,P\,\}$ amounts to checking

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

In other words, we need to check the equivalence of two loop-free programs. As program equivalence is undecidable in general, we introduce a syntactic fragment of loop-free cpGCL programs for which program equivalence is decidable. The syntax and semantics of this fragment, called cReDiP–conditional Rectangular Discrete Probabilistic programs, are described in Table 3.

Let us explain the important parts of cReDiP in detail: The word "rectangular" in cReDiP emphasizes that Boolean guards can only be of the form $x < n$ where $n \in \mathbb{N}$ is a constant. With $G_{x<n}$, we denote the generating function $G$ restricted to the terms with low enough order, thus satisfying the guard $x < n$. By nesting of if-statements, axis-aligned hyper-rectangles can still be identified, i.e., we can express conjunction, disjunction and negation of guards. The latter suffices to only consider observe(false) statements, as again we can reconstruct the full "rectangular" expressiveness for observe statements. Since our program variables cannot take negative values, we define x−− by $\max(x-1,0)$.

Intuitively, the statement $x \mathrel{+}= \texttt{iid}\,(D,y)$ can be interpreted as a special kind of loop, namely $\texttt{loop}(y)\{x \mathrel{+}= \texttt{sample}(D)\}$ where the number of iterations is given by program variable $y$. More specifically, $x \mathrel{+}= \texttt{iid}\,(D,y)$ combines a series of operations: First independently sample $y$ many random variables from distribution $D$ and second, sum up the sampled values and increment $x$ by that amount. For example, the program $P \coloneqq y \coloneqq 10; x \coloneqq 0; x \mathrel{+}= \texttt{iid}\,(\texttt{bernoulli}\,(1/2)\,,y)$ describes a binomial distribution in $X$ with parameters $n = 10$ and $p = 1/2$, i.e. $\llbracket P\rrbracket = Y^{10}\cdot(1/2{+}1/2X)^{10}$.

Table 3. The *unconditioned* semantics for cReDiP programs.

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

Note that cReDiP is as expressive as cpGCL is (both are Turing complete), however their loop-free fragments differ. The latter implies, that we can decide the equivalence of loop-free cReDiP programs using the following approach based on extended second order PGFs [Chen et al. 2022a].

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

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

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

Due to the PGF semantics being an instance of the general framework of Kozen's measure transformer semantics [Klinkenberg et al. 2020; Kozen 1981], the posterior distribution of a cReDiP program is uniquely determined by its semantics on all possible Dirac distributions. Thus, for the purpose of deciding program equivalence,

$$\hat{G} := (1 - X_1 U_1)^{-1} \cdots (1 - X_k U_k)^{-1} = \sum\nolimits_{\sigma \in \mathbb{N}^k} \mathbf{X}^\sigma \mathbf{U}^\sigma = 1 + (1\mathbf{X})\mathbf{U} + (1\mathbf{X}^2)\mathbf{U}^2 + \ldots \in \mathbb{R}[[\mathbf{X}, \mathbf{U}]]$$

is of particular importance, as it enumerates all possible point-mass distributions for $\mathbf{X}$. The meta-indeterminates $\mathbf{U}$ thereby serve the purpose of "remembering" the initial state. Note that $\hat{G}$ does not encode any observe violation probabilities as they can be immediately removed from the equivalence check, which is formalized by the following Lemma 11.

**Lemma 11 (Error Term Pass-Through).** *For every program* $P$ *and every* $F \in$ eFPS,

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

Lemma 11 asserts that the error term $[\lightning]_F X_{\lightning}$ passes through the transformer unaffected. However, eSOP can be embedded into eFPS by adding the additional indeterminates $\mathbf{U}$. This identification allows us to apply the denotational semantics $\llbracket \cdot \rrbracket$ also on eSOP elements.

**Theorem 12 (eSOP Semantics).** *Let* $P$ *be a loop-free* cReDiP *Program. Let* $G = \sum_{\sigma \in \mathbb{N}^k} G_\sigma \mathbf{U}^\sigma \in$ eSOP. *The* eSOP *semantics of* $P$ *is*

$$\llbracket P \rrbracket(G) = \sum\nolimits_{\sigma \in \mathbb{N}^k} \llbracket P \rrbracket(G_\sigma) \cdot \mathbf{U}^\sigma \, .$$

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

Prog. 6. A truncated geometric distribution generator.

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

Prog. 7. A loop-free cReDiP invariant of Prog. 6.

Since the loop-free cReDiP semantics is well-defined on eSOP, we can simultaneously compute posterior distributions for multiple given input states. Therefore, consider the example program $x := x+1$ together with input $G = XU+X^2U^2+X^3U^3$. Applying $[\![P]\!](G)$ yields $X^2U+X^3U^2+X^4U^3 = X \cdot G$. We have computed all posterior distributions for initial states $X = 1, X = 2, X = 3$ in one shot. Generalizing this idea, we can equivalently characterize program equivalence of loop-free programs using eSOP.

**Lemma 13** (eSOP **Characterization).** *Let $P_1$ and $P_2$ be loop-free* cReDiP *programs with $Vars(P_i) \subseteq \{x_1, \ldots, x_k\}$ for $i \in \{1, 2\}$. Further, consider a vector $\mathbf{U} = (U_1, \ldots, U_k)$ of meta indeterminates, and let $\hat{G}$ be the eSOP $(1 - X_1U_1)^{-1} \cdots (1 - X_kU_k)^{-1} \in \mathbb{R}[[\mathbf{X}, \mathbf{U}]]$. Then,*

$$\forall G \in \text{eFPS}.\ [\![P_1]\!](G) \ = \ [\![P_2]\!](G) \qquad \Longleftrightarrow \qquad [\![P_1]\!](\hat{G}) \ = \ [\![P_2]\!](\hat{G}).$$

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

**Corollary 14 (Decidability of Equivalence).** *Let $P, Q$ be two loop-free* cReDiP *programs. Then,*

$$\forall G \in \text{eFPS}.\ [\![P]\!](G) = [\![Q]\!](G) \quad \textit{is decidable.}$$

Proof. By utilizing Lemma 13, we can rephrase the problem of determining program equivalence through the eSOP characterization $[\![P_1]\!](\hat{G}) = [\![P_2]\!](\hat{G})$. It is worth noting that $\hat{G}$ represents a *rational closed-form* eSOP $\hat{G} = \frac{1}{1-X_1U_1} \frac{1}{1-X_2U_2} \cdots \frac{1}{1-X_kU_k} \in \mathbb{R}[[\mathbf{X}, \mathbf{U}]]$ . For our purposes, we can disregard the portion of $\hat{G}$ that describes the initial observe violation behavior, as it immediately cancels out (see Lemma 13). As $\hat{G}$ is in rational closed form, both $[\![P_1]\!](\hat{G})$ and $[\![P_2]\!](\hat{G})$ must also possess a rational closed form (since cReDiP semantics preserve closed forms; see Table 3 and [Chen et al. 2022a]). Additionally, the effective computation of $[\![P_1]\!](\hat{G}) = F_1/H_1$ and $[\![P_2]\!](\hat{G}) = F_2/H_2$ is possible because both $P_1$ and $P_2$ are assumed to be loop-free programs.

In $\mathbb{R}[[\mathbf{X}, X_{\lightning}, \mathbf{U}]]$, the question of whether two eFPS represented as rational closed forms, namely $F_1/H_1$ and $F_2/H_2$, are equal can be decided. This is determined by the equation

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

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

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

**Theorem 15.** *Let $L =$* `while ( B ) { P }` *$\in$* cReDiP *be UAST with loop-free body $P$ and $I$ be a loop-free* cReDiP *program. It is decidable whether $[\![L]\!] = [\![I]\!]$.*

Proof. The correctness is an immediate consequence of Theorem 9 and Corollary 14. □

We demonstrate the invariant reasoning technique by Example 16.

**Example 16 (Geometric Distribution Generator).** Prog. 6 describes an iterative algorithm that repeatedly flips a fair coin – while counting the number of trials – until seeing heads, and observes that the number of trails is less than 3. Assume we want to compute the posterior distribution for input $1 \cdot Y^1 X^0$ (i.e. $y = 1$ and $x = 0$). We first evaluate lfp $\Phi_{B,P}$. Using Theorem 9 (2), we perform an *equivalence check* on the invariant in Prog. 7. As Prog. 6 and 7 are equivalent, we substitute the loop-free program for the `while`-loop and continue. The resulting posterior distribution for input $Y$ is $[\![P]\!](Y) = \frac{4}{7} + \frac{2}{7}X + \frac{1}{7}X^2$. Since Prog. 6 is UAST, this is the *precise posterior distribution*. The step-by-step computation of the equivalence check can be found in Appendix E. ◁

To summarize, *reasoning about program equivalence using eSOPs enables exact Bayesian inference for* `cReDiP` *programs containing loops*. Our notion of equivalence describes exact equivalence for the *unconditioned* semantics, i.e., the `while`-loop and the loop-free invariant generate the same distributions and `observe` violation probabilities. Given these circumstances, it is immediately clear that also the normalized distributions are equivalent $[\![L]\!] = [\![I]\!] \implies \mathit{norm}([\![L]\!]) = \mathit{norm}([\![I]\!])$.

### 4.3 Equivalence of Conditioned Semantics

Ideally, we aim to have a weaker notion of equivalence between programs $P$ and $Q$, i.e.,

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

and express this in terms of eSOPs. First, we lift the operator *norm* to the eSOP domain:

**Definition 17 (Conditioning on eSOP).** *Let $G \in \mathsf{eSOP}$. The function*

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

*is called the* conditioning function.

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

**Proposition 18.** *Let $F_1, F_2 \in \mathsf{ePGF}$, with $p \coloneqq [\lightning]_{F_1} = [\lightning]_{F_2}$. Then,*

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

Intuitively, in that case *cond* behaves kind of linear as it satisfies additivity. Generalizing this concept to a finite amount of equal `observe`-violation properties we get the following.

**Corollary 19 (Partitioning).** *Let $S$ be a finite partitioning of $\mathbb{N}^k = S_1 \uplus \cdots \uplus S_m$ with $[\lightning]_{G_\sigma} = [\lightning]_{G_{\sigma'}}$, for all $\sigma, \sigma' \in S_i,\ 1 \le i \le m$. Then:*

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

*where $[\lightning]_{S_i}$ denotes the observation-violation probability in $S_i$. For eSOPs satisfying this property, we can compute cond by*

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

```
/// $(1 - XU)^{-1}(1 - YV)^{-1}$
$x := 0$⨟
/// $(1 - U)^{-1}(1 - YV)^{-1}$
$x$ += iid (bernoulli ($1/2$) , $y$)⨟
/// $2(1 - U)^{-1}(2 - (1 + X)YV)^{-1}$
$y := 0$⨟
/// $2(1 - U)^{-1}(2 - (1 + X)V)^{-1}$
observe ( $x < 1$ )⨟
/// $\frac{2(1 - X_{\lightning})}{(1 - U)(2 - V)} + \frac{X_{\lightning}}{(1 - U)(1 - V)}$
/// $\sum_{i=0}^{\infty} \frac{(2^{-i} + (1 - 2^{-i})X_{\lightning})\, V^i}{(1 - U)}$
```

Prog. 8. Program with infinitely many observe violation probabilities.

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

Prog. 9. $n$-geometric generator with success probability $q/3$ for $0 \leq q \leq 3$.

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

Prog. 10. $n$-geometric invariant with parameter $p$.

Unfortunately, there exist already loop-free programs for which a finite partitioning is impossible. An example is provided in Prog. 8. Given an initial distribution for variable $y$, the program computes the sum of $y$-many independent and identically distributed Bernoulli variables with success probability $1/2$. This is equivalent to sampling from a binomial distribution with $y$ trials and probability $1/2$. Finally, it marginalizes the distribution by assigning $y$ to zero and conditions on the event that $x$ is less than 1, resulting in $\sum_{i=0}^{\infty} \frac{(2^{-i}+(1-2^{-i})X_{\lightning})V^i}{(1-U)}$. We can read off that for any initial state $(x, y)$ we obtain a *different* observe violation probability $(1 - 2^{-y})$, hence we cannot finitely partition the state space into equal violation probability classes. There is another challenge when considering the equivalence of normalized distributions: Evaluating *cond* on (closed-form) eSOPs yields that $\mathit{cond}(\llbracket P \rrbracket(\hat{G})) = \mathit{cond}(\llbracket Q \rrbracket(\hat{G}))$. This further implies $\forall \sigma \in \mathbb{N}^k.\ \mathit{norm}(\llbracket P \rrbracket(\mathbf{X}^{\sigma})) = \mathit{norm}(\llbracket Q \rrbracket(\mathbf{X}^{\sigma}))$, i.e., equivalence on point-mass distributions. However, we do not necessarily have the precise equivalence as per Eq. (5), because the *norm* operator used to define *cond* is a non-linear function[7] and thus the point-mass distributions cannot be combined in a sensible way. However, in many use cases we are only interested in the behavior of a specific initial state where such a result on point-mass equivalence can still be useful.

## 5 FINDING INVARIANTS USING PARAMETER SYNTHESIS

In contrast to the previous section which aims at *validating a given invariant*, we address the problem of *finding* such invariants. To recall, the invariant synthesis problem is stated as follows: Given a while-loop $L$, find a loop-free cReDiP program $I$ such that $\Phi_{B,P}(\llbracket I \rrbracket) = \llbracket I \rrbracket$. Similar to classical programs, synthesizing invariants for probabilistic programs is hard. For related problems, e.g., finding invariants in terms of weakest preexpectations, there exist sound and complete synthesis algorithms for *subclasses* of loops and properties that can be verified by piecewise linear templates [Batz et al. 2023]. We, in turn, leverage the power of eSOPs to achieve decidability results for a subclass of invariant candidates. Invariants can be in parametric form, e.g., described by the program $I_p = \{\, x := 1 \,\}\ [\, p \,]\ \{\, x := 0 \,\}$ modeling a Bernoulli distribution with symbolic parameter

[7] For the unconditioned semantics, general equivalence $\llbracket P \rrbracket = \llbracket Q \rrbracket$ follows from the linearity of the transformer.

$p$. Sometimes, the general shape of an invariant program is derivable from the loop $L$, but the precise parameters are tricky to tune. We illustrate the idea by Example 20.

**Example 20 ($n$-Geometric Parameter Synthesis).** Prog. 9 is a variant of Prog. 2, where instead of requiring one success (setting $h = 0$), we need $n$ successes to terminate. Furthermore, the individual success probability is now $\frac{q}{3}$, where $q$ is a symbolic parameter. It seems natural that this program might encode the $n$-fold geometric distribution[8] with individual success probability $\frac{q}{3}$. This suggests to formulate the invariant template $Q_p$ given in Prog. 10, where $c$ is a sum of $n$ geometric distributions with an unknown parameter $p$. Using Theorem 9, we can derive the equivalence of Prog. 9 and Prog. 10 and obtain an equation in $p$ and $q$:

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

$$\llbracket Q_p \rrbracket(\hat{G}) = -\frac{(-1 + C - pC)}{(-1 + CV)(-1 + C - pC + pU)}$$

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

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

This approach works in general as the following theorem describes:

**Theorem 21 (Decidability of Parameter Synthesis).** *Let $W$ be a* cReDiP while *loop and $I_{\mathbf{p}}$ be a parametrized loop-free* cReDiP *program. It is decidable whether there exist parameter values $\rho$ such that the instantiated template $I_\rho$ is an invariant, i.e.,*

$$\exists\, \mathbf{p} \in \mathbb{R}^l. \quad \llbracket W \rrbracket \;=\; \llbracket I_{\mathbf{p}} \rrbracket\,.$$

Proof. The proof is a variant of Corollary 14. Full details are provided in Appendix D. □

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

**Example 22 (Hypergeometric Invariant).** Prog. 11 encodes a biased 2-dimensional bounded random walk. In each turn, it decrements one of the variables with equal probability until either the value of $m$ or $n$ arrives at 0. For any fixed program state $(0, 0) \neq (m, n) \in \mathbb{N}^2$, the number of loop iterations is bounded by $n + m - 1$. We are interested in the exact posterior distribution for arbitrary input distributions. Due to its finite nature for any particular input distribution with finite support, we can analyze this program automatically using Prodigy by unfolding the loop $m + n - 1$ times. For instance, the resulting distribution for an initial Dirac distribution describing the state $(a, b)$, is $\llbracket P \rrbracket(M^a N^b) = \sum_{i=1}^{a} \frac{M^i}{2^{a+b-i}} \cdot \binom{a+b-i-1}{b-1} + \sum_{i=1}^{b} \frac{N^i}{2^{a+b-i}} \cdot \binom{a+b-i-1}{a-1}$. Using the simplification function in Mathematica [Inc. 2023], we derive the closed form,

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

Prog. 11. Dependent negative binomial variables.

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

[8] Sometimes also called negative binomial distribution.

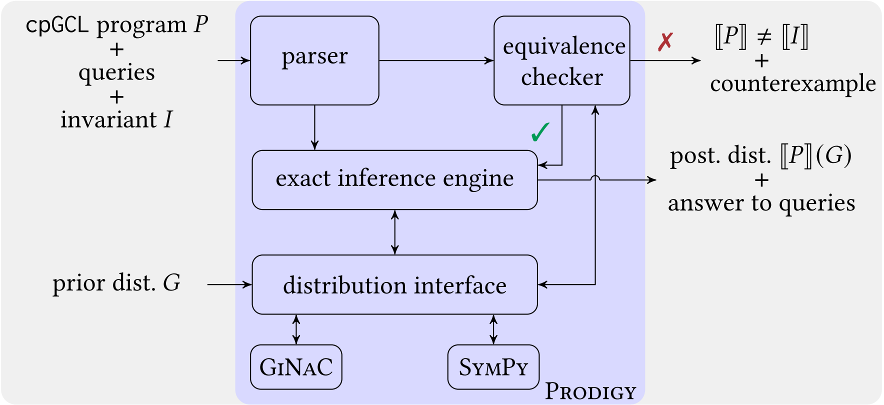

cpGCL program $P$ + queries + invariant $I$
parser
equivalence checker
✗
$\llbracket P \rrbracket \neq \llbracket I \rrbracket$ + counterexample
✓
exact inference engine
post. dist. $\llbracket P \rrbracket(G)$ + answer to queries
prior dist. $G$
distribution interface
GiNaC
SymPy
Prodigy

Fig. 3. A sketch of the Prodigy workflow.

Here ${}_2F_1$ denotes the hypergeometric function[9]. It shows that the distribution is in some sense linked to the hypergeometric distribution, indicated by the ${}_2F_1$ terms. Even though that function is quite complex, taking derivatives in $M$ or $N$ respectively is straightforward, i.e., $\frac{\partial}{\partial x}{}_2F_1(p_1, p_2, p_3; x) = \frac{p_1 p_2}{c}{}_2F_1(p_1+1, p_2+1, p_3+1; M)$. Thus, extracting many properties of interest can still be computed exactly using the closed-form expression. It is unknown (to us) whether some loop-free cReDiP invariant program generates this closed-form distribution. However, the GF semantics enables us to prove that the precise semantics of Prog. 11 is captured by checking $\forall a, b \in \mathbb{N}.\ (a, b) \neq (0, 0) \implies I(a, b) = \Phi_{B,P}(I)(a, b)$, combined with the fact that it universally certainly terminates.

## 6 EMPIRICAL EVALUATION OF PRODIGY

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

### 6.1 Implementation of Prodigy

Prodigy implements exact inference for cpGCL programs; its high-level structure is depicted in Figure 3. Given a cpGCL program $P$ (optionally with queries to the output distribution, e.g., expected values, tail bounds and moments) together with a prior distribution $G$, Prodigy parses the program, performs PGF-based distribution transformations (via the inference engine), and finally outputs the posterior distribution $\llbracket P \rrbracket(G)$ (plus answers to the queries, if any). For the distribution transformation, Prodigy implements an internal distribution interface acting as an abstract datatype for probability distributions in the form of formal power series. Such an abstraction allows for an easy integration of alternative distribution representations (not necessarily related to generating functions) and various computer algebra systems (CAS) in the backend (Prodigy currently supports SymPy [Meurer et al. 2017] and GiNaC [Bauer et al. 2002; Vollinga 2006]). When (UAST) loops are encountered, Prodigy asks for a user-provided invariant $I$ and then performs the equivalence check such that it can either infer the output distribution or conclude that $\llbracket P \rrbracket \neq \llbracket I \rrbracket$ while providing counterexamples. In the absence of an invariant, Prodigy is capable of computing

[9] More about this closed form and algorithms to compute closed forms alike can be found in [Petkovsek et al. 1996].
[10] https://github.com/LKlinke/Prodigy

Table 4. Exact inference results for loopy probabilistic programs (some with parameter synthesis, marked by _param); timings are given in seconds.

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

under-approximations of the posterior distribution by unfolding the loop up to a specified accuracy or number of loop unrollings.

## 6.2 Benchmarks

We collected a set of 37 benchmarks, 16 of them related to inferring distributions for loopy programs. This set consists of examples provided by $\lambda$-PSI [Gehr et al. 2020], Genfer [Zaiser et al. 2023], and Prodigy. All experiments were evaluated on MacOS Sonoma 14.0 with a 2,4 GHz Quad-Core Intel Core i5 and 16GB RAM. For each benchmark, we run Prodigy with both CAS backends, i.e., SymPy and GiNaC. For loop-free benchmarks, Prodigy is compared against $\lambda$-PSI[11] and Genfer[12] – the two closest tools (among those in Section 8). As Prodigy is an exact inference engine, all tools are run using *exact arithmetic*. The initial prior distribution is 1 which means all variables are initialized to 0 with probability 1 and no `observe`-violations have occurred. All timings are averaged over 20 iterations per benchmark and we measured the time used for performing inference (computing the posterior distribution). The experiments aim to answer questions in terms of (1) *Effectiveness:* Can Prodigy effectively do exact inference on the selected benchmarks, including equivalence checking and invariant synthesis for programs with loops? (2) *Efficiency:* How does Prodigy compare to the most related tools? How do the CAS backends SymPy and GiNaC compare to each other?

## 6.3 Experimental Results

*General observations.* Our approach is capable of computing posterior distributions for a variety of programs in less than 0.1 seconds. For loop-free benchmarks, *exact* Bayesian inference based on generating functions (Genfer, Prodigy) performs better than $\lambda$-PSI on *discrete* probabilistic

[11] We used the commit `9db68ba9581b7a1211f1514e44e7927af24bd398`.
[12] We used the commit `5911de13f16bc3c28703f1631c5c4847f9ebac9a`.

programs with Genfer being the fastest in most instances. Prodigy is the only tool that is able to deal with unbounded loopy programs.

*Results for loop-free programs.* Whereas we are primarily interested in programs featuring unbounded loops, we compared Prodigy to $\lambda$-PSI and Genfer for loop-free benchmarks (see Table 5 in Appendix E). Our experiments show that Genfer can be up to two orders of magnitude faster. They achieve this by computing Taylor approximations up to a maximal degree, which in some cases is sufficient as the posterior distribution has just finite support. However, for programs admitting a compact symbolic representation of the posterior distribution, like `lin_regession_unbiased`, Prodigy is faster; as in this case, manipulating closed forms is more efficient than computing Taylor series. Moreover, Genfer is unable to deal with non-linear observations as in the `pi` benchmark. The same holds for instances where symbolic parameters are involved. Prodigy outperforms the symbolic engine of $\lambda$-PSI on almost every instance whilst Prodigy has a comparable performance to the dynamic programming strategy of $\lambda$-PSI.

*Results for loopy programs.* Table 4 depicts the empirical results for loopy programs. The column Program lists the benchmarks. The columns SymPy and GiNaC report their run-times in seconds when used as backend of Prodigy. The timing in boldface marks the fastest variant. As these benchmarks all include loops, they are not supported by $\lambda$-PSI and Genfer.

Recall that reasoning about loops involves an equivalence check against a user-specified invariant program. Finding the right invariant (if it exists in the loop-free `cReDiP` fragment) is intricate. We support the user in discovering such invariants by allowing symbolic parameters for distributions, e.g., one can write `geom` $(p)$ where $p$ is a symbolic parameter. For benchmarks subject to parameter synthesis, we also provide the anticipated parameter constraints (or values) inferred automatically by Prodigy. Whenever this is the case, we point out that for the GiNaC timings, discharging the resulting equation systems is achieved using SymPy solvers, which is due to the missing functionality of GiNaC to solve these equation systems. Overall, GiNaC is faster than SymPy by a factor of 100, as is similar to the loop-free benchmarks.

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

**Example 23 (Bounded Retransmission Protocol).** Prog. 12 describes a conditioned variant of the bounded retransmission protocol (BRP) [Batz et al. 2023; D'Argenio et al. 2001] which attempts to transmit $s$ packets over a lossy channel, where each individual packet gets lost with probability 1%. The transmission is considered successful, if *none* of the packets needs more than 4 retransmission tries. Additionally, we observe that all but the last 9 packets are received successfully. Fig. 4 illustrates the protocol as Markov chain. Notice that the number of packets to be sent is parametrized by the (possibly infinite-support) initial distribution of $s$ – modeling an *infinite family* of finite-state Markov chains – and hence renders techniques like probabilistic model checking [Katoen 2016] infeasible. Provided with a suitable invariant (cf. Appendix E) with parameter $p$ in the probabilities, Prodigy is able to infer that, with $p = 10^{-10}$, Prog. 12 is equivalent to this (loop-free) invariant, thereby yielding the exact posterior distribution (for any initial distribution of $s$ with rational closed-form) in the form of a PGF. From this PGF, we can derive, e.g., when starting with $s \sim$ `geom` $(1/2)$, the *transmission-failure probability* of BRP, i.e., the probability that Prog. 12 terminates with $f > 4$ is around $9.9789 \times 10^{-11}$ (see the precise probability in Appendix E).

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

Prog. 12. A conditioned variant of BRP.

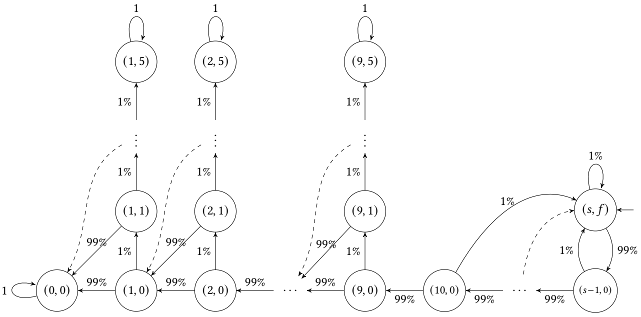


Fig. 4. The Markov chain illustrating Prog. 12.

## 7 LIMITATIONS OF EXACT INFERENCE USING EFPS

We discuss some limitations of the presented inference approach considering guard evaluations, non-rational probabilities and scalability. Prog. 14 models a variant of the famous Collatz algorithm [Andrei and Masalagiu 1998]. The Collatz conjecture states that for all positive integers $m$ there exists $n \in \mathbb{N}$ such that for the Collatz function $C(m) \coloneqq n/2$ for $n \equiv (0 \bmod 2)$ and $3n + 1$ otherwise; the $n$-th fold iteration of the function is $C^n(m) = 1$. We have adapted the program syntax slightly and make use of the `loop` statement to represent the $n$-fold repetition of a code block. The program basically behaves as the usual Collatz function with the only exception that in the case where a number is divisible by two, we have a small chance not dividing $x$ by 2 but instead executing the `else` branch. Note that the instruction $x \equiv_2 0(\text{mod } 2)$ still preserves rational closed forms as we can compute its semantics by $\frac{F(\mathrm{X})+F(-\mathrm{X})}{2}$. When analyzing the run-times of the program we observe surprising results: for $(n = 1)$ we obtain a result in 0.010631 seconds; $(n = 2)$ is computed in 0.049891 seconds and for $(n = 3)$ it suddenly increases to 88.689832 seconds. We think that this phenomenon arises from the fact that evaluating expressions like $x \equiv 0 \pmod 2$ repeatedly, gets increasingly difficult as it is implemented in Prodigy by means of arithmetic progressions.

Another challenge is guard evaluation, i.e., filtering out the corresponding terms of a formal power series. In case we are interested in the relation between two variables (like $x = y$) when both have marginal distributions with infinite support, Prodigy cannot compute the result. As an approximation heuristic it computes under-approximations of the *exact* posterior distribution. Note that if either $x$ or $y$ has a finite-support marginal distribution, the posterior can be computed by enumeration. An interesting example why one cannot even strive for such a potential closed-form operation preserving rational closed-forms is Prog. 13. For this program, its variable $r$ evaluates to 1 with *non-rational*, not even algebraic probability $1/\pi$ after termination [Flajolet et al. 2011] – thus beyond `cReDiP` capabilities. It thus might be interesting what syntactic restrictions *exactly capture rational* closed forms.

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

## 8 RELATED WORK

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

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

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

Prog. 13. Non-algebraic Probabilities.

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

Prog. 14. Probabilistic Collatz's.

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

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

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

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

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

As an alternative to PGFs, many probabilistic systems employ *probability density function* (PDF) representations of distributions, e.g., ($\lambda$)PSI [Gehr et al. 2016, 2020], AQUA [Huang et al. 2021] and Hakaru [Narayanan et al. 2016], as well as the density compiler in [Bhat et al. 2012, 2017]. These systems are dedicated to inference for programs encoding joint (discrete-)continuous distributions with conditioning. Reasoning about the underlying PDF representations, however, amounts to resolving complex integral expressions in order to answer inference queries. Furthermore, ($\lambda$)PSI admits *only bounded looping behaviors*. Dice [Holtzen et al. 2020] employs weighted model counting to enable potentially scalable exact inference for discrete probabilistic programs, yet is also confined to statically bounded loops. Stein and Staton [2021] proposed a denotational semantics based on Markov categories for continuous probabilistic programs with exact conditioning and bounded looping behaviors. A similar direction is taken by Bichsel et al. [2018]. They investigate the connections between `observe`-violations, non-termination, and errors raised by, e.g., division by zero; their semantics is based on Markov kernels. A recently proposed language PERPL [Chiang et al. 2023] compiles probabilistic programs with unbounded recursion into systems of polynomial equations and solves them directly for least fixed points using numerical methods. A related approach by Stuhlmüller and Goodman [2012] uses dynamic programming techniques transforming probabilistic programs with unbounded recursion into factored sum-product networks, i.e., a particular way of representing an equation system. However, this technique cannot handle infinite-support distributions. The tool Mora [Bartocci et al. 2020a,b] supports exact inference for various types of Bayesian networks, but relies on a restricted form of intermediate representation known as prob-solvable loops, whose behaviors can be expressed by a system of C-finite recurrences admitting closed-form solutions.

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

## 9 CONCLUSION

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

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

## ACKNOWLEDGMENTS

This research work has been partially funded by the ERC Advanced Project FRAPPANT under grant No. 787914, by the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 101008233, and by the ZJU Education Foundation's Qizhen Talent program. The authors would like to thank Leo Mommers for his assistance in producing the benchmark results and his work on part of the implementation.

## REFERENCES

Samson Abramsky and Achim Jung. 1994. Domain Theory. In *Handbook of Logic in Computer Science, vol. 3: Semantic Structures*. Clarendon Press.

Nathanael L. Ackerman, Cameron E. Freer, and Daniel M. Roy. 2019. On the Computability of Conditional Probability. *J. ACM* 66, 3 (2019).

Ştefan Andrei and Cristian Masalagiu. 1998. About the Collatz conjecture. *Acta Informatica* 35, 2 (1998), 167–179.

Christel Baier, Joachim Klein, Linda Leuschner, David Parker, and Sascha Wunderlich. 2017. Ensuring the Reliability of Your Model Checker: Interval Iteration for Markov Decision Processes. In *CAV (2) (LNCS, Vol. 10426)*. Springer, 160–180.

Jialu Bao, Nitesh Trivedi, Drashti Pathak, Justin Hsu, and Subhajit Roy. 2022. Data-Driven Invariant Learning for Probabilistic Programs. In *CAV (1) (LNCS, Vol. 13371)*. Springer, 33–54.

Gilles Barthe, Thomas Espitau, Luis María Ferrer Fioriti, and Justin Hsu. 2016. Synthesizing Probabilistic Invariants via Doob's Decomposition. In *CAV (1) (LNCS, Vol. 9779)*. Springer, 43–61.

Gilles Barthe, Joost-Pieter Katoen, and Alexandra Silva (Eds.). 2020. *Foundations of Probabilistic Programming*. Cambridge University Press.

Ezio Bartocci, Laura Kovács, and Miroslav Stankovic. 2019. Automatic Generation of Moment-Based Invariants for Prob-Solvable Loops. In *ATVA (LNCS, Vol. 11781)*. Springer, 255–276.

Ezio Bartocci, Laura Kovács, and Miroslav Stankovic. 2020a. Analysis of Bayesian Networks via Prob-Solvable Loops. In *ICTAC (LNCS, Vol. 12545)*. Springer, 221–241.

Ezio Bartocci, Laura Kovács, and Miroslav Stankovic. 2020b. Mora - Automatic Generation of Moment-Based Invariants. In *TACAS (1) (LNCS, Vol. 12078)*. Springer, 492–498.

Kevin Batz, Mingshuai Chen, Sebastian Junges, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. 2023. Probabilistic Program Verification via Inductive Synthesis of Inductive Invariants. In *TACAS (2) (LNCS, Vol. 13994)*. Springer, 410–429.

Kevin Batz, Mingshuai Chen, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Philipp Schröer. 2021. Latticed $k$-Induction with an Application to Probabilistic Programs. In *CAV (2) (LNCS, Vol. 12760)*. Springer, 524–549.

Kevin Batz, Sebastian Junges, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Philipp Schröer. 2020. PrIC3: Property Directed Reachability for MDPs. In *CAV (2) (LNCS, Vol. 12225)*. Springer, 512–538.

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

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

Sooraj Bhat, Johannes Borgström, Andrew D. Gordon, and Claudio V. Russo. 2017. Deriving Probability Density Functions from Probabilistic Functional Programs. *Log. Methods Comput. Sci.* 13, 2 (2017).

Benjamin Bichsel, Timon Gehr, and Martin T. Vechev. 2018. Fine-Grained Semantics for Probabilistic Programs. In *ESOP (LNCS, Vol. 10801)*. Springer, 145–185.

Olivier Bournez and Florent Garnier. 2005. Proving Positive Almost-Sure Termination. In *RTA (LNCS, Vol. 3467)*. Springer, 323–337.

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

Bob F Caviness and Jeremy R Johnson. 2012. *Quantifier Elimination and Cylindrical Algebraic Decomposition*. Springer Science & Business Media.

Milan Češka, Christian Dehnert, Nils Jansen, Sebastian Junges, and Joost-Pieter Katoen. 2019. Model Repair Revamped – On the Automated Synthesis of Markov Chains. In *From Reactive Systems to Cyber-Physical Systems (LNCS, Vol. 11500)*. Springer, 107–125.

Aleksandar Chakarov and Sriram Sankaranarayanan. 2013. Probabilistic Program Analysis with Martingales. In *CAV (LNCS, Vol. 8044)*. Springer, 511–526.

Aleksandar Chakarov and Sriram Sankaranarayanan. 2014. Expectation Invariants for Probabilistic Program Loops as Fixed Points. In *SAS (LNCS, Vol. 8723)*. Springer, 85–100.

Krishnendu Chatterjee, Hongfei Fu, and Amir Kafshdar Goharshady. 2016. Termination Analysis of Probabilistic Programs Through Positivstellensatz's. In *CAV (1) (LNCS, Vol. 9779)*. Springer, 3–22.

Krishnendu Chatterjee, Hongfei Fu, and Petr Novotný. 2020. Termination Analysis of Probabilistic Programs with Martingales. In *Foundations of Probabilistic Programming*, Gilles Barthe, Joost-Pieter Katoen, and Alexandra Silva (Eds.). Cambridge University Press, 221–258.

Krishnendu Chatterjee, Petr Novotný, and Dorde Zikelic. 2017. Stochastic Invariants for Probabilistic Termination. In *POPL*. ACM, 145–160.

Mingshuai Chen, Joost-Pieter Katoen, Lutz Klinkenberg, and Tobias Winkler. 2022a. Does a Program Yield the Right Distribution? Verifying Probabilistic Programs via Generating Functions. In *CAV (1) (LNCS, Vol. 13371)*. Springer, 79–101.

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

Yu-Fang Chen, Chih-Duo Hong, Bow-Yaw Wang, and Lijun Zhang. 2015. Counterexample-Guided Polynomial Loop Invariant Generation by Lagrange Interpolation. In *CAV (1) (LNCS, Vol. 9206)*. Springer, 658–674.

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

Gregory F. Cooper. 1990. The Computational Complexity of Probabilistic Inference Using Bayesian Belief Networks. *Artif. Intell.* 42, 2-3 (1990), 393–405.

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

Pedro R. D'Argenio, Bertrand Jeannet, Henrik Ejersbo Jensen, and Kim Guldstrand Larsen. 2001. Reachability Analysis of Probabilistic Systems by Successive Refinements. In *PAPM-PROBMIV (Lecture Notes in Computer Science, Vol. 2165)*. Springer, 39–56.

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

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

Yijun Feng, Lijun Zhang, David N. Jansen, Naijun Zhan, and Bican Xia. 2017. Finding Polynomial Loop Invariants for Probabilistic Programs. In *ATVA (LNCS, Vol. 10482)*. Springer, 400–416.

Philippe Flajolet, Maryse Pelletier, and Michèle Soria. 2011. On Buffon Machines and Numbers. In *SODA*. SIAM, 172–183.

Philippe Flajolet and Robert Sedgewick. 2009. *Analytic Combinatorics*. Cambridge University Press.

Daniel J. Fremont, Edward Kim, Tommaso Dreossi, Shromona Ghosh, Xiangyu Yue, Alberto L. Sangiovanni-Vincentelli, and Sanjit A. Seshia. 2022. Scenic: A Language for Scenario Specification and Data Generation. *Machine Learning Journal* (2022).

Timon Gehr, Sasa Misailovic, and Martin T. Vechev. 2016. PSI: Exact Symbolic Inference for Probabilistic Programs. In *CAV (1) (LNCS, Vol. 9779)*. Springer, 62–83.

Timon Gehr, Samuel Steffen, and Martin T. Vechev. 2020. $\lambda$PSI: Exact Inference for Higher-Order Probabilistic Programs. In *PLDI*. ACM, 883–897.

Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. 2014. Probabilistic Programming. In *FOSE*. ACM, 167–181.

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

Friedrich Gretz, Joost-Pieter Katoen, and Annabelle McIver. 2013. Prinsys - On a Quest for Probabilistic Loop Invariants. In *QEST (LNCS, Vol. 8054)*. Springer, 193–208.

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

Arnd Hartmanns and Benjamin Lucien Kaminski. 2020. Optimistic Value Iteration. In *CAV (2) (LNCS, Vol. 12225)*. Springer, 488–511.

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

Zixin Huang, Saikat Dutta, and Sasa Misailovic. 2021. AQUA: Automated Quantized Inference for Probabilistic Programs. In *ATVA (LNCS, Vol. 12971)*. Springer, 229–246.

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

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

Nils Jansen, Christian Dehnert, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Lukas Westhofen. 2016. Bounded Model Checking for Probabilistic Programs. In *ATVA (LNCS, Vol. 9938)*. 68–85.

Norman L Johnson, Adrienne W Kemp, and Samuel Kotz. 2005. *Univariate Discrete Distributions*. Vol. 444. John Wiley & Sons.

Benjamin Lucien Kaminski. 2019. *Advanced weakest precondition calculi for probabilistic programs*. Ph. D. Dissertation. RWTH Aachen University.

Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. 2019. On the Hardness of Analyzing Probabilistic Programs. *Acta Inform.* 56, 3 (2019), 255–285.

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

Joost-Pieter Katoen. 2016. The Probabilistic Model Checking Landscape. In *LICS*. ACM, 31–45.

Joost-Pieter Katoen, Annabelle McIver, Larissa Meinicke, and Carroll C. Morgan. 2010. Linear-Invariant Generation for Probabilistic Programs: Automated Support for Proof-Based Methods. In *SAS (LNCS, Vol. 6337)*. Springer, 390–406.

Lutz Klinkenberg, Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Joshua Moerman, and Tobias Winkler. 2020. Generating Functions for Probabilistic Programs. In *LOPSTR (LNCS, Vol. 12561)*. Springer, 231–248.

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

Dexter Kozen. 1981. Semantics of Probabilistic Programs. *J. Comput. Syst. Sci.* 22, 3 (1981), 328–350.

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

Jean-Louis Lassez, V. L. Nguyen, and Liz Sonenberg. 1982. Fixed Point Theorems and Semantics: A Folk Tale. *Inf. Process. Lett.* 14, 3 (1982), 112–116.

Michael L. Littman, Judy Goldsmith, and Martin Mundhenk. 1998. The Computational Complexity of Probabilistic Planning. *J. Artif. Intell. Res.* 9 (1998), 1–36.

Annabelle McIver and Carroll Morgan. 2005. *Abstraction, Refinement and Proof for Probabilistic Systems.* Springer.

Nicholas Metropolis and Stanisław Ulam. 1949. The Monte Carlo Method. *J. Am. Stat. Assoc.* 44, 247 (1949), 335–341.

Aaron Meurer et al. 2017. SymPy: Symbolic computing in Python. *PeerJ Comput. Sci.* 3 (2017), e103.

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

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

Michael Mitzenmacher and Eli Upfal. 2005. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis.* Cambridge University Press.

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

Praveen Narayanan, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov. 2016. Probabilistic Inference by Program Transformation in Hakaru (System Description). In *FLOPS (LNCS, Vol. 9613)*. Springer, 62–79.

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

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

Joël Ouaknine and James Worrell. 2014. On the Positivity Problem for Simple Linear Recurrence Sequences. In *ICALP (2) (LNCS, Vol. 8573)*. Springer, 318–329.

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

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

Tim Quatmann and Joost-Pieter Katoen. 2018. Sound Value Iteration. In *CAV (1) (LNCS, Vol. 10981)*. Springer, 643–661.

Dan Roth. 1996. On the Hardness of Approximate Reasoning. *Artif. Intell.* 82, 1 (1996), 273–302.

Nasser Saheb-Djahromi. 1978. Probabilistic LCF. In *MFCS (LNCS, Vol. 64)*. Springer, 442–451.

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

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

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

Dario Stein and Sam Staton. 2021. Compositional Semantics for Probabilistic Programs with Exact Conditioning. In *LICS*. IEEE, 1–13.

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

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

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

Jens Vollinga. 2006. GiNaC–Symbolic Computation with C++. *Nucl. Instrum. Methods Phys. Res.* 559, 1 (2006), 282–284.

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

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

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

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

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

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

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

# APPENDIX

## A DOMAIN THEORY

**Notation.** The set of natural numbers, including 0 is denoted by $\mathbb{N}$. $\mathbb{R}_{\geq 0}^{\infty}$ denotes the set of non-negative real numbers extended by $\infty$, whereby the ordering relation is conservative with $\infty \geq r$ for all $r \in \mathbb{R}_{\geq 0}^{\infty}$. For any sets $D$ and $D'$, we write $(D \to D')$ as the set of functions $\{f : D \to D'\}$. We write vectors in bold-face notations like $\mathbf{X}$ for $(X_1, \ldots, X_k)$ and $\mathbf{1} = (1, \ldots, 1)$ where the dimension is clear from the context. We sometimes use Lambda calculus notations describing anonymous functions, e.g. we write $\lambda x.\, x^2$ for a function that maps $x \mapsto x^2$. Multivariate partial derivatives are compactly denoted by $\partial_x^i f \coloneqq \frac{\partial^i f}{\partial x^i}$.

**Definition 24 (Partial Order).** *A* partial order $(D, \sqsubseteq)$ *is a set* $D$ *along with a binary relation* $\sqsubseteq\, \subseteq (D \times D)$ *fulfilling the following properties:*

*(1) Reflexivity:* $\forall d \in D.\, d \sqsubseteq d$.
*(2) Antisymmetry:* $\forall d, d' \in D.\, d \sqsubseteq d' \wedge d' \sqsubseteq d \implies d = d'$.
*(3) Transitivity:* $\forall d, d', d'' \in D.\, d \sqsubseteq d' \wedge d' \sqsubseteq d'' \implies d \sqsubseteq d''$.

**Definition 25 (Complete Lattice).** *A* complete lattice *is a partial order* $(D, \sqsubseteq)$ *such that every subset* $S \subseteq D$ *has a* supremum *denoted by* $\sup S$ *(sometimes also* $\bigsqcup S$*). An element of the domain* $D$ *is called an* upper bound *of* $S$ *if and only if* $\forall s \in S.\, s \sqsubseteq d$*. Further,* $d$ *is the* least *upper bound of* $S$ *if and only if* $d \sqsubseteq d'$ *for every upper bound* $d'$ *of* $S$*.*

**Definition 26 (Monotonic Function).** *Let* $(D, \sqsubseteq)$ *and* $(D', \sqsubseteq')$ *be complete lattices. A function* $f : D \to D'$ *is* monotonic *if and only if:*

$$\forall d, d' \in D.\, d \sqsubseteq d' \qquad \implies \qquad f(d) \sqsubseteq' f(d')\ .$$

**Definition 27 (Continuous Functions).** *Let* $(D, \sqsubseteq)$ *and* $(D', \sqsubseteq')$ *be complete lattices. A function* $f : D \to D'$ *is* Scott-continuous *if and only if for every* $\omega$*-chain, i.e., every set* $S = \{s_n \mid n \in \mathbb{N}\} \subseteq D$ *such that* $s_0 \sqsubseteq s_1 \sqsubseteq s_2 \sqsubseteq \ldots$*, it holds that:*

$$\sup\{f(s) \mid s \in S\} \quad = \quad f(\sup S)\ .$$

**Lemma 28 (Continuous Functions are Monotone).** *Let* $(D, \sqsubseteq)$ *and* $(D', \sqsubseteq')$ *be complete lattices, and* $f : D \to D'$ *be a continuous function. Then* $f$ *is monotonic.*

Proof. Let $d, d' \in D$ such that $d \sqsubseteq d'$.

$$\begin{aligned} & d \sqsubseteq d' \\ \implies & \sup\{d, d'\} = d' \\ \implies & \sup\{f(d), f(d')\} = f(\sup\{d, d'\}) = f(d') && \text{(Scott-Cont. of } f\text{)} \\ \implies & f(d) \sqsubseteq' \sup\{f(d), f(d')\} = f(d') && \square \end{aligned}$$

**Lemma 29 (Lifting of Partial Orders).** *Let* $(D', \leq')$ *be a partial order and let* $\sqsubseteq'$ *be a point-wise lifting of* $\leq'$*, i.e., for an arbitrary domain* $D$ *and any* $f, g \in (D \to D')$*,* $f \sqsubseteq' g$ *if and only if* $\forall d \in D.\, f(d) \leq' g(d)$*. Then,* $(D \to D', \sqsubseteq')$ *is a partial order.*

Proof. Let $f, g, h \in (D \to D')$. We need to show that $\sqsubseteq$ is a partial order, i.e., it is reflexive, antisymmetric and transitive.

$$\begin{aligned}
\textbf{Reflexivity}: \quad & && \forall d \in D.\ f(d) \preceq' f(d) && \text{(refl. of } <'\text{)} \\
& && \Longrightarrow f \sqsubseteq' f && \\
\textbf{Transitivity}: \quad & && f \sqsubseteq' h \text{ and } h \sqsubseteq' g && \\
& && \Longrightarrow \forall d \in D.\ f(d) \preceq' h(d) \text{ and } h(d) \preceq' g(d) && \\
& && \Longrightarrow \forall d \in D.\ f(d) \preceq' g(d) && \text{(trans. of } \preceq'\text{)} \\
& && \Longrightarrow f \sqsubseteq g' && \\
\textbf{Antisymmetry}: \quad & && f \sqsubseteq' g \text{ and } g \sqsubseteq' g && \\
& && \Longrightarrow \forall d \in D.\ f(d) \preceq' g(d) \text{ and } g(d) \preceq' f(d) && \\
& && \Longrightarrow \forall d \in D.\ f(d) = g(d) && \text{(antisym. of } \preceq'\text{)} \\
& && \Longrightarrow f = g && \square
\end{aligned}$$

**Lemma 30 (Point-Wise Lifting of Complete Lattices).** *Let $(D', \preceq')$ be a complete lattice and $\sqsubseteq'$ be a point-wise lifting of $\preceq'$, i.e. for an arbitrary domain $D$ and any $f, g \in (D \to D')$, let $f \sqsubseteq' g$ if and only if $\forall f \in D.\ f(d) \preceq' g(d)$. Then $(D \to D', \sqsubseteq')$ is a complete lattice.*

Proof. We claim that every subset $S \subseteq (D \to D')$ has a least upper bound given by

$$\sup S = \lambda d.\, \sup S_d, \quad \text{where} \quad S_d \coloneqq \{f(d) \mid f \in S\} \subseteq D' .$$

First we show that $\sup S$ is an upper bound, as for every $f \in S$

$$\begin{aligned}
& \forall d \in D. \quad f(d) \preceq' \sup S_d = (\sup S)(d) \\
\Longrightarrow\ & f \sqsubseteq' \sup S
\end{aligned}$$

Second, $\sup S$ is the *least* upper bound. Therefore, let $\hat{f}$ be an upper bound of $S$.

$$\begin{aligned}
& \forall f \in S. \quad f \sqsubseteq' \hat{f} \\
\Longrightarrow\ & \forall f \in S. \quad \forall d \in D.\ f(d) \preceq' \hat{f}(d) \\
\Longrightarrow\ & \forall d \in D. \quad (\sup S)(d) = \sup S_d \preceq' \hat{f}(d) \\
\Longrightarrow\ & \sup S \sqsubseteq' f && \square
\end{aligned}$$

**Theorem 31 (Fixed Point Theorems** [Abramsky and Jung 1994; Lassez et al. 1982]**).** *Let $f\colon D \to D$ be a continuous function on a complete lattice $(D, \sqsubseteq)$. Then $f$ possesses a least fixed point denoted* lfp $f$*, which is given by:*

*(1)* $\operatorname{lfp} f = \sup\{f^n(\bot) \mid n \in \mathbb{N}\}$, *where $f^n$ denotes the $n$-fold application of $f$, and $\bot = \sup \emptyset$ is the least element of $D$.*
*(2)* $\operatorname{lfp} f = \inf\{d \in D \mid f(d) \sqsubseteq d\}$.

## B SEMANTICS USING EFPS

**Corollary 32 (Partial Orders over eFPS).** $(\mathrm{eFPS}, \preceq)$ *as well as the point-wise lifting on functions* $(\mathrm{eFPS} \to \mathrm{eFPS}, \sqsubseteq)$ *are partial orders.*

Proof. Consider the coefficient function $[\cdot]_F \in (\mathbb{N}^k \cup \{\lightning\} \to \mathbb{R}_{\geq 0}^{\infty})$ which uniquely determines the eFPS $F$. We think of the order $\preceq$ as acting on the domain $(\mathbb{N}^k \cup \{\lightning\} \to \mathbb{R}_{\geq 0}^{\infty})$. Thus, $\preceq$ can be interpreted as the point-wise lifting of the (total) order $\leq$ on $\mathbb{R}_{\geq 0}^{\infty}$, i.e., $(\mathrm{eFPS}, \preceq)$ is a partial order by applying Lemma 29. Since $\sqsubseteq$ is a point-wise lifting of $\preceq$, we can argue analogously for $(\mathrm{eFPS} \rightarrow \mathrm{eFPS}, \sqsubseteq)$. □

**Corollary 33 (Complete Lattices over eFPS).** *Both partial orders* $(\mathrm{eFPS}, \preceq)$ *and* $(\mathrm{eFPS} \rightarrow \mathrm{eFPS}, \sqsubseteq)$ *are complete lattices.*

Proof. Analogously to the proof of Corollary 32 we note that $\preceq$ is a point-wise lifting of $\leq$ on $\mathbb{R}_{\geq 0}^{\infty}$, and $\sqsubseteq$ is a point-wise lifting on $\preceq$. Therefore applying Lemma 30 twice yields the claimed result. □

**Lemma 34 (Continuity of $\Phi_{B,P}$).** *Let $P$ be a cpGCL program and let $B$ be a Boolean guard. The characteristic functional $\Phi_{B,P}$ is continuous on the domain* $(\mathrm{eFPS} \rightarrow \mathrm{eFPS}, \sqsubseteq)$.

Proof.

$$\begin{aligned}
\Phi_{B,P}(\sup S) &= \Phi_{B,P}(\lambda F.\ \sup \{\psi(F) \mid \psi \in S\}) \\
&= \lambda F.\ [\lightning]_F \cdot X_{\lightning} + \langle F \rangle_{\neg B} \\
&\qquad + (\lambda F.\ \sup \{\psi(F) \mid \psi \in S\})(\llbracket P \rrbracket(\langle F \rangle_B)) && (\text{Def. } \Phi_{B,P}) \\
&= \lambda F.\ [\lightning]_F \cdot X_{\lightning} + \langle F \rangle_{\neg B} + \sup \left\{\psi(\llbracket P \rrbracket(\langle F \rangle_B)) \mid \psi \in S\right\} && (\text{Evaluate inner } \lambda\text{-function}) \\
&= \lambda F.\ \sup \left\{[\lightning]_F \cdot X_{\lightning} + \langle F \rangle_{\neg B} + \psi(\llbracket P \rrbracket(\langle F \rangle_B)) \mid \psi \in S\right\} && (\text{Include constants in sup}) \\
&= \sup \left\{\lambda F.\ [\lightning]_F \cdot X_{\lightning} + \langle F \rangle_{\neg B} + \psi(\llbracket P \rrbracket(\langle F \rangle_B)) \mid \psi \in S\right\} && (\text{sup defined point-wise}) \\
&= \sup \left\{\Phi_{B,P}(\psi) \mid \psi \in S\right\} && (\text{Def. } \Phi_{B,P})
\end{aligned}$$

□

**Lemma 35 (Continuity of Auxiliary Functions).** *For all $\sigma \in \mathbb{N}^k \cup \{\lightning\}$ and Boolean guards $B$, the following functions are continuous:*

*(1) the coefficient function $[\sigma]$*
*(2) the restriction $\langle \cdot \rangle_B$*
*(3) the mass $|\cdot|$*

Proof. 1 and 2 follow directly from the coefficient-wise definition of sup on eFPS. For 3, let $S = \{F_i \mid i \in \mathbb{N}\} \subseteq \mathrm{eFPS}$ be an $\omega$-chain with $F_0 \preceq F_1 \preceq F_2 \preceq \ldots$. Then:

$$\begin{aligned}
|\sup S| &= \sum_{\sigma \in \mathbb{N}^k} [\sigma]_{\sup S} \\
&= \sum_{\sigma \in \mathbb{N}^k} \sup \left\{[\sigma]_{F_i} \mid i \in \mathbb{N}\right\} \\
&= \sup \left\{ \sum_{\sigma \in \mathbb{N}^k} [\sigma]_{F_i} \;\middle|\; i \in \mathbb{N} \right\} && (\text{Monotone Convergence Theorem}) \\
&= \sup \{|F_i| \mid i \in \mathbb{N}\}
\end{aligned}$$

□

**Theorem 36 (Continuity of $\llbracket \cdot \rrbracket$).** *For every cpGCL program $P$, $\llbracket P \rrbracket$ is continuous on the domain* $(\mathrm{eFPS} \rightarrow \mathrm{eFPS})$.

Proof. Let $S \subseteq \mathrm{eFPS}$. The proof proceeds by induction over the structure of $P$:

*Case $P = \mathtt{skip}$:*

$$\llbracket P \rrbracket(\sup S) = \sup S = \sup \{F \mid F \in S\} = \sup \left\{\llbracket P \rrbracket(F) \mid F \in S\right\}$$

*Case* $P = x_i \coloneqq E$*:*

$$\begin{aligned}
\llbracket P \rrbracket(\sup S) &= \llbracket P \rrbracket \left( [\lightning]_{\sup S} \cdot X_{\lightning} + \sum_{\sigma \in \mathbb{N}^k} [\sigma]_{\sup S} \cdot \mathbf{X}^{\sigma} \right) \\
&= [\lightning]_{\sup S} \cdot X_{\lightning} + \sum_{\sigma \in \mathbb{N}^k} [\sigma]_{\sup S} \cdot X_1^{\sigma_1} \cdots X_i^{E(\sigma)} \cdots X_k^{\sigma_k} \\
&= \sup_{F \in S} \left\{ [\lightning]_F \cdot X_{\lightning} + \sum_{\sigma \in \mathbb{N}^k} [\sigma]_F \cdot X_1^{\sigma_1} \cdots X_i^{E(\sigma)} \cdots X_k^{\sigma_k} \right\} \\
&= \sup_{F \in S} \left\{ \llbracket P \rrbracket(F) \right\}
\end{aligned}$$

*Case* $P =$ `observe` $( B )$*:*

$$\begin{aligned}
\llbracket P \rrbracket(\sup S) &= \big( [\lightning]_{\sup S} + |\langle \sup S \rangle_{\neg B}| \big) \cdot X_{\lightning} + \langle \sup S \rangle_B \\
&= \big( [\lightning]_{\sup S} + \sup \{ |\langle F \rangle_{\neg B}| \mid F \in S \} \big) \cdot X_{\lightning} \\
&\qquad + \sup \{ \langle F \rangle_B \mid F \in S \} && \text{(Cont. of } |\cdot|, \langle \cdot \rangle_B\text{)} \\
&= \sup \big\{ ( [\lightning]_F + |\langle F \rangle_{\neg B}| ) \cdot X_{\lightning} + \langle F \rangle_B \big\} \\
&= \sup \big\{ \llbracket P \rrbracket(F) \mid F \in S \big\}
\end{aligned}$$

*Case* $P = \{ P_1 \} \; [ p ] \; \{ P_2 \}$*:*

$$\begin{aligned}
\llbracket P \rrbracket(\sup S) &= p \cdot \llbracket P_1 \rrbracket(\sup S) + (1 - p) \cdot \llbracket P_2 \rrbracket(\sup S) \\
&= p \cdot \sup \big\{ \llbracket P_1 \rrbracket(F) \mid F \in S \big\} \\
&\qquad + (1 - p) \cdot \sup \big\{ \llbracket P_2 \rrbracket(F) \mid F \in S \big\} && \text{(I.H. on } P_1 \text{ and } P_2\text{)} \\
&= \sup \big\{ p \cdot \llbracket P_1 \rrbracket(F) + (1 - p) \cdot \llbracket P_2 \rrbracket(F) \mid F \in S \big\} \\
&= \sup \big\{ \llbracket P \rrbracket(F) \mid F \in S \big\}
\end{aligned}$$

*Case* $P =$ `if` $( B ) \; \{ P_1 \}$ `else` $\{ P_2 \}$*:*

$$\begin{aligned}
\llbracket P \rrbracket(\sup S) &= [\lightning]_{\sup S} \cdot X_{\lightning} + \llbracket P_1 \rrbracket(\langle \sup S \rangle_P) + \llbracket P_2 \rrbracket(\langle \sup S \rangle_{\neg B}) \\
&= [\lightning]_{\sup S} \cdot X_{\lightning} + \sup \big\{ \llbracket P_1 \rrbracket(\langle F \rangle_P) \mid F \in S \big\} \\
&\qquad + \sup \big\{ \llbracket P_2 \rrbracket(\langle F \rangle_{\neg B}) \mid F \in S \big\} && \text{(I.H. on } P_1 \text{ and } P_2\text{)} \\
&= \sup \big\{ [\lightning]_F \cdot X_{\lightning} + \llbracket P_1 \rrbracket(\langle F \rangle_P) + \llbracket P_2 \rrbracket(\langle F \rangle_{\neg B} \mid F \in S \big\} \\
&= \sup \big\{ \llbracket P \rrbracket(F) \mid F \in S \big\}
\end{aligned}$$

*Case* $P = P_1 \fatsemi P_2$*:*

$$\begin{aligned}
\llbracket P \rrbracket(\sup S) &= \llbracket P_2 \rrbracket \big( \llbracket P_1 \rrbracket(\sup S) \big) \\
&= \llbracket P_2 \rrbracket \big( \sup \big\{ \llbracket P_1 \rrbracket(F) \mid F \in S \big\} \big) && \text{(I.H. on } P_1\text{)} \\
&= \sup \big\{ \llbracket P_2 \rrbracket(\llbracket P_1 \rrbracket(F)) \mid F \in S \big\} && \text{(I.H. on } P_2\text{)} \\
&= \sup \big\{ \llbracket P \rrbracket(F) \mid F \in S \big\}
\end{aligned}$$

*Case* $P =$ `while` $( B ) \; \{ P_1 \}$*:* In this case, we use that for all $n \in \mathbb{N}$, $\Phi^n_{B,P_1}(\bot)$ is continuous, which we prove by induction:

*Base case: $n = 0$.*

$$\Phi^0_{B,P_1}(\bot)(\sup S) = 0 = \sup\left\{\Phi^0_{B,P_1}(\bot)(F) \mid F \in S\right\}$$

*Induction step:*

$$\begin{aligned}
\Phi^{n+1}_{B,P_1}(\bot)(\sup S) &= \Phi_{B,P_1}\left(\Phi^n_{B,P_1}(\bot)\right)(\sup S) \\
&= [\lightning]_{\sup S} + \langle \sup S\rangle_{\neg B} \\
&\qquad + \Phi^n_{B,P_1}(\bot)(⟦P_1⟧(\langle \sup S\rangle_B)) && \text{(Def. } \Phi_{B,P_1}\text{)} \\
&= [\lightning]_{\sup S} + \sup\{\langle F\rangle_{\neg B} \mid F \in S\} \\
&\qquad + \Phi^n_{B,P_1}(\bot)\left(\sup\left\{⟦P_1⟧(\langle F\rangle_B) \mid F \in S\right\}\right) && \text{(Cont. of } \langle\,\cdot\,\rangle_B\text{, outer I.H. on } P_1\text{)} \\
&= [\lightning]_{\sup S} + \sup\{\langle F\rangle_{\neg B} \mid F \in S\} \\
&\quad + \sup\left\{\Phi^n_{B,P_1}(\bot)\left(⟦P_1⟧(\langle F\rangle_B)\right) \mid F \in S\right\} && \text{(Inner I.H.)} \\
&= \sup_{F\in S}\left\{[\lightning]_F + \langle F\rangle_{\neg B} + \Phi^n_{B,P_1}(\bot)\left(⟦P_1⟧(\langle F\rangle_B)\right)\right\} \\
&= \sup\left\{\Phi^{n+1}_{B,P_1}(\bot)(F) \mid F \in S\right\}
\end{aligned}$$

With this, it follows:

$$\begin{aligned}
⟦P⟧(\sup S) &= \left(\sup_{n\in\mathbb{N}} \Phi^n_{B,P_1}(\bot)\right)(\sup S) \\
&= \sup\left\{\Phi^n_{B,P_1}(\bot)(\sup S) \mid n \in \mathbb{N}\right\} \\
&= \sup\left\{\sup\left\{\Phi^n_{B,P_1}(\bot)(F) \mid F \in S\right\} \mid n \in \mathbb{N}\right\} && \text{(Cont. of } \Phi^n_{B,P_1}(\bot)\text{)} \\
&= \sup\left\{\sup\left\{\Phi^n_{B,P_1}(\bot)(F) \mid n \in \mathbb{N}\right\} \mid F \in S\right\} && \text{(swap suprema)} \\
&= \sup\left\{\sup\left\{\Phi^n_{B,P_1}(\bot) \mid n \in \mathbb{N}\right\}(F) \mid F \in S\right\} \\
&= \sup\left\{⟦P⟧(F) \mid F \in S\right\}
\end{aligned}$$

□

**Lemma 37 (Linearity of Auxiliary Functions).** *For all $\sigma \in \mathbb{N}^k \cup \{\lightning\}$, $\alpha \in \mathbb{R}^\infty_{\geq 0}$, $F, G \in$ eFPS and Boolean guards $B$, the following functions are linear:*

*(1) The coefficient function $[\sigma]$, i.e. $[\sigma]_{\alpha F+G} = \alpha \cdot [\sigma]_F + [\sigma]_G$.*
*(2) The restriction $\langle\,\cdot\,\rangle_B$, i.e. $\langle \alpha F + G\rangle_B = \alpha \cdot \langle F\rangle_B + \langle G\rangle_B$.*
*(3) The mass $|\cdot|$, i.e. $|\alpha F + G| = \alpha \cdot |F| + |G|$.*

Proof. (1) follows from coefficient-wise addition and scalar multiplication on eFPS:

$$\alpha F + G = \alpha \cdot \left([\lightning]_F \cdot X_{\lightning} + \sum_{\sigma\in\mathbb{N}^k} [\sigma]_F \cdot \mathbf{X}^\sigma\right) \tag{6}$$

$$+ \left([\lightning]_G \cdot X_{\lightning} + \sum_{\sigma\in\mathbb{N}^k} [\sigma]_G \cdot \mathbf{X}^\sigma\right)$$

$$= (\alpha \cdot [\lightning]_F + [\lightning]_G) \cdot X_{\lightning} + \sum_{\sigma\in\mathbb{N}^k} (\alpha \cdot [\sigma]_F + [\sigma]_G) \cdot \mathbf{X}^\sigma \tag{7}$$

By Definition 3:

$$\alpha F + G = [\lightning]_{\alpha F+G} \cdot X_{\lightning} + \sum_{\sigma \in \mathbb{N}^k} [\sigma]_{\alpha F+G} \cdot \mathbf{X}^{\sigma} \tag{8}$$

Comparing coefficients of Eq. (7) and Eq. (8) yields $[\sigma]_{\alpha F+G} = \alpha \cdot [\sigma]_F + [\sigma]_G$ for all $\sigma \in \mathbb{N}^k \cup \{\lightning\}$.

(2) Using the result of 1:

$$\begin{aligned} \langle \alpha F + G \rangle_B &= \sum_{\sigma \models B} [\sigma]_{\alpha F+G} \cdot \mathbf{X}^{\sigma} \\ &= \sum_{\sigma \models B} (\alpha \cdot [\sigma]_F + [\sigma]_G) \cdot \mathbf{X}^{\sigma} \\ &= \alpha \cdot \sum_{\sigma \models B} [\sigma]_F \cdot \mathbf{X}^{\sigma} + \sum_{\sigma \models B} [\sigma]_G \cdot \mathbf{X}^{\sigma} \\ &= \alpha \cdot \langle F \rangle_B + \langle G \rangle_B \end{aligned}$$

(3) follows directly from the linearity of variable substitution:

$$|\alpha F + G| = (\alpha F + G)(\mathbf{1}) = \alpha \cdot F(\mathbf{1}) + G(\mathbf{1}) = \alpha \cdot |F| + |G|$$

□

**Lemma 38 ($\Phi_{B,P}$ Preserves Linearity).** *Let $\psi \colon \mathsf{eFPS} \to \mathsf{eFPS}$ be a linear function. If $[\![P]\!]$ is linear, then $\Phi_{B,P}(\psi)$ is linear as well.*

Proof.

$$\begin{aligned} &\Phi_{B,P}(\psi)(\alpha F + G) \\ =\ &\big(\lambda F.\ [\lightning]_F X_{\lightning} + \langle F \rangle_{\neg B} + \psi([\![P]\!](\langle F \rangle_B))\big)\,(\alpha F + G) \\ =\ &[\lightning]_{\alpha F+G} X_{\lightning} + \langle \alpha F + G \rangle_{\neg B} + \psi([\![P]\!](\langle \alpha F + G \rangle_B)) \\ =\ &(\alpha[\lightning]_F + [\lightning]_G) \cdot X_{\lightning} + \alpha\langle F \rangle_{\neg B} + \langle G \rangle_{\neg B} + \psi([\![P]\!](\alpha\langle F \rangle_B + \langle G \rangle_B)) \qquad \text{(Lin. of } \langle\,\cdot\,\rangle_B \text{ (Lemma 37))} \\ =\ &(\alpha[\lightning]_F + [\lightning]_G) \cdot X_{\lightning} + \alpha\langle F \rangle_{\neg B} + \langle G \rangle_{\neg B} \\ &\qquad + \psi(\alpha[\![P]\!](\langle F \rangle_B) + [\![P]\!](\langle G \rangle_B)) \qquad \text{(Lin. of } [\![P]\!]) \\ =\ &(\alpha[\lightning]_F + [\lightning]_G) \cdot X_{\lightning} + \alpha\langle F \rangle_{\neg B} + \langle G \rangle_{\neg B} \\ &\qquad + \alpha \cdot \psi([\![P]\!](\langle F \rangle_B)) + \psi([\![P]\!](\langle G \rangle_B)) \qquad \text{(Lin. of } \psi) \\ =\ &\alpha \cdot \big([\lightning]_F X_{\lightning} + \langle F \rangle_{\neg B} + \psi([\![P]\!](\langle F \rangle_B))\big) \\ &\qquad + \big([\lightning]_G X_{\lightning} + \langle G \rangle_{\neg B} + \psi([\![P]\!](\langle G \rangle_B))\big) \\ =\ &\alpha \cdot \Phi_{B,P}(\psi)(F) + \Phi_{B,P}(\psi)(G) \end{aligned}$$

□

**Corollary 39.** *If $[\![P]\!]$ is linear, then $\Phi^n_{B,P}(\bot)$ is linear for all $n \in \mathbb{N}$, i.e.*

$$\Phi^n_{B,P}(\bot)(\alpha F + G) = \alpha \cdot \Phi^n_{B,P}(\bot)(F) + \Phi^n_{B,P}(\bot)(G).$$

Proof. By induction:

*Base case: $n = 0$.* $\Phi^0_{B,P}(\bot) = \bot$ is linear, as $\bot(\alpha F + G) = 0 = \alpha \cdot \bot(F) + \bot(G)$.

*Induction step:* By the induction hypothesis $\Phi^n_{B,P}(\bot)$ is a linear function. Therefore, $\Phi^{n+1}_{B,P}(\bot) = \Phi_{B,P}(\Phi^n_{B,P}(\bot))$ is also linear by Lemma 38. □

**Theorem 40 (Linearity of $[\![\cdot]\!]$).** *The semantics transformer $[\![\cdot]\!]$ is linear, i.e. for any cpGCL program $P$*

$$[\![P]\!](\alpha F + G) = \alpha \cdot [\![P]\!](F) + [\![P]\!](G).$$

Proof. By induction over the structure of $P$:

*Case $P = \texttt{skip}$:*

$$\begin{aligned} [\![P]\!](\alpha F + G) &= \alpha F + G \\ &= \alpha[\![P]\!](F) + [\![P]\!](G) \end{aligned}$$

*Case $P = x_i \coloneqq E$:*

$$\begin{aligned} &[\![P]\!](\alpha F + G) \\ &= [\![P]\!]\big((\alpha[\lightning]_F + [\lightning]_G)X_\lightning \\ &\qquad + \sum_{\sigma\in\mathbb{N}^k} (\alpha[\sigma]_F + [\sigma]_G)\mathbf{X}^\sigma\big) \\ &= (\alpha[\lightning]_F + [\lightning]_G)X_\lightning \\ &\qquad + \sum_{\sigma\in\mathbb{N}^k} (\alpha[\sigma]_F + [\sigma]_G)X_1^{\sigma_1}\cdots X_i^{E(\sigma)}\cdots X_k^{\sigma_k} \\ &= \alpha\cdot\left([\lightning]_F X_\lightning + \sum_{\sigma\in\mathbb{N}^k}[\sigma]_F X_1^{\sigma_1}\cdots X_i^{E(\sigma)}\cdots X_k^{\sigma_k}\right) \\ &\qquad + \left([\lightning]_G X_\lightning + \sum_{\sigma\in\mathbb{N}^k}[\sigma]_G X_1^{\sigma_1}\cdots X_i^{E(\sigma)}\cdots X_k^{\sigma_k}\right) \\ &= \alpha\cdot[\![P]\!](F) + [\![P]\!](G) \end{aligned}$$

*Case $P = \texttt{observe (}\, B\, \texttt{)}$:*

$$\begin{aligned} &[\![P]\!](\alpha F + G) \\ &= (\alpha[\lightning]_F + [\lightning]_G + |\langle \alpha F + G\rangle_{\neg B}|)X_\lightning + \langle \alpha F + G\rangle_B \\ &= (\alpha[\lightning]_F + [\lightning]_G + \alpha\,|\langle F\rangle_{\neg B}| + |\langle G\rangle_{\neg B}|)X_\lightning \\ &\qquad + \alpha\langle F\rangle_B + \langle G\rangle && \text{(Lin. of } \langle\,\cdot\,\rangle_B \text{ (Lemma 37))} \\ &= \alpha\left(([\lightning]_F + |\langle F\rangle_{\neg B}|)X_\lightning + \langle F\rangle_B\right) \\ &\quad + \left(([\lightning]_G + |\langle G\rangle_{\neg B}|)X_\lightning + \langle G\rangle_B\right) \\ &= \alpha[\![P]\!](F) + [\![P]\!](G) \end{aligned}$$

*Case* $P = \{\, P_1 \,\}\ [\, p \,]\ \{\, P_2 \,\}$*:*

$$
\begin{aligned}
&\llbracket P \rrbracket(\alpha F + G) \\
&= p \cdot \llbracket P_1 \rrbracket(\alpha F + G) + (1-p) \cdot \llbracket P_2 \rrbracket(\alpha F + G) \\
&= p \cdot (\alpha \llbracket P_1 \rrbracket(F) + \llbracket P_1 \rrbracket(G)) + (1-p) \cdot (\alpha \llbracket P_2 \rrbracket(F) + \llbracket P_2 \rrbracket(G)) && \text{(I.H.)} \\
&= \alpha \left(p \cdot \llbracket P_1 \rrbracket(F) + (1-p) \cdot \llbracket P_2 \rrbracket(F)\right) \\
&\qquad + \left(p \cdot \llbracket P_1 \rrbracket(G) + (1-p) \cdot \llbracket P_2 \rrbracket(G)\right) \\
&= \alpha \llbracket P \rrbracket(F) + \llbracket P \rrbracket(G)
\end{aligned}
$$

*Case* $P = \texttt{if}\ (\, B \,)\ \{\, P_1 \,\}\ \texttt{else}\ \{\, P_2 \,\}$*:*

$$
\begin{aligned}
&\llbracket P \rrbracket(\alpha F + G) \\
&= (\alpha [\lightning]_F + [\lightning]_G) X_{\lightning} + \llbracket P_1 \rrbracket(\langle \alpha F + G \rangle_B) + \llbracket P_2 \rrbracket(\langle \alpha F + G \rangle_{\neg B}) \\
&= (\alpha [\lightning]_F + [\lightning]_G) X_{\lightning} + \llbracket P_1 \rrbracket(\alpha \langle F \rangle_B + \langle G \rangle_B) \\
&\qquad + \llbracket P_2 \rrbracket(\alpha \langle F \rangle_{\neg B} + \langle G \rangle_{\neg B}) && \text{(Lin. of } \langle \cdot \rangle_B \text{ (Lemma 37))} \\
&= (\alpha [\lightning]_F + [\lightning]_G) X_{\lightning} + \alpha \llbracket P_1 \rrbracket(\langle F \rangle_B) + \llbracket P_1 \rrbracket(\langle G \rangle_B) \\
&\qquad + \alpha \llbracket P_2 \rrbracket(\langle F \rangle_{\neg B}) + \llbracket P_2 \rrbracket(\langle G \rangle_{\neg B}) && \text{(I.H.)} \\
&= \alpha \left([\lightning]_F X_{\lightning} + \llbracket P_1 \rrbracket(\langle F \rangle_B) + \llbracket P_2 \rrbracket(\langle F \rangle_{\neg B}\right) \\
&\qquad + \left([\lightning]_G X_{\lightning} + \llbracket P_1 \rrbracket(\langle G \rangle_B) + \llbracket P_2 \rrbracket(\langle G \rangle_{\neg B}\right) \\
&= \alpha \llbracket P \rrbracket(F) + \llbracket P \rrbracket(G)
\end{aligned}
$$

*Case* $P = P_1 \fatsemi P_2$*:*

$$
\begin{aligned}
&\llbracket P \rrbracket(\alpha F + G) \\
&= \llbracket P_2 \rrbracket(\llbracket P_1 \rrbracket(\alpha F + G)) \\
&= \llbracket P_2 \rrbracket(\alpha \llbracket P_1 \rrbracket(F) + \llbracket P_1 \rrbracket(G)) && \text{(I.H.)} \\
&= \alpha \llbracket P_2 \rrbracket(\llbracket P_1 \rrbracket(F)) + \llbracket P_2 \rrbracket(\llbracket P_1 \rrbracket(G)) && \text{(I.H.)} \\
&= \alpha \llbracket P \rrbracket(F) + \llbracket P \rrbracket(G)
\end{aligned}
$$

*Case* $P = \texttt{while}\,(\, B \,)\,\{\, P_1 \,\}$*:*

$$
\begin{aligned}
&\llbracket P \rrbracket(\alpha F + G) \\
&= (\mathrm{lfp}\ \Phi_{B,P_1})(\alpha F + G) \\
&= \left(\sup \left\{\Phi^n_{B,P_1}(\bot) \mid n \in \mathbb{N}\right\}\right)(\alpha F + G) \\
&= \ \sup \left\{\Phi^n_{B,P_1}(\bot)(\alpha F + G) \mid n \in \mathbb{N}\right\} \\
&= \ \sup \left\{\alpha \cdot \Phi^n_{B,P_1}(\bot)(F) + \Phi^n_{B,P_1}(\bot)(G) \mid n \in \mathbb{N}\right\} && \text{(Corollary 39, } \llbracket P_1 \rrbracket \text{ lin. by I.H.)} \\
&= \alpha \cdot \sup \left\{\Phi^n_{B,P_1}(\bot)(F) \mid n \in \mathbb{N}\right\} + \sup \left\{\Phi^n_{B,P_1}(\bot)(G) \mid n \in \mathbb{N}\right\} \\
&= \alpha \cdot \left(\sup \left\{\Phi^n_{B,P_1}(\bot) \mid n \in \mathbb{N}\right\}\right)(F) + \left(\sup \left\{\Phi^n_{B,P_1}(\bot) \mid n \in \mathbb{N}\right\}\right)(G) \\
&= \alpha \cdot (\mathrm{lfp}\ \Phi_{B,P_1})(F) + (\mathrm{lfp}\ \Phi_{B,P_1})(G) \\
&= \alpha \llbracket P \rrbracket(F) + \llbracket P \rrbracket(G)
\end{aligned}
$$

□

**Lemma 41 (Error Term Pass-Through).** *For every program $P$ and every $F \in \mathsf{eFPS}$, the error term $[\lightning]_F X_\lightning$ passes through the transformer unaffected, i.e.*

$$\llbracket P \rrbracket(F) = \llbracket P \rrbracket\left(\sum_{\sigma \in \mathbb{N}^k} [\sigma]_F \mathbf{X}^\sigma\right) + [\lightning]_F X_\lightning \,.$$

Proof. By linearity of $\llbracket P \rrbracket$, we get:

$$\begin{aligned}\llbracket P \rrbracket(F) &= \llbracket P \rrbracket\left(\sum_{\sigma \in \mathbb{N}^k} [\sigma]_F \mathbf{X}^\sigma + [\lightning]_F X_\lightning\right)\\ &= \llbracket P \rrbracket\left(\sum_{\sigma \in \mathbb{N}^k} [\sigma]_F \mathbf{X}^\sigma\right) + [\lightning]_F \cdot \llbracket P \rrbracket(X_\lightning)\end{aligned}$$

It therefore remains to be shown that $\llbracket P \rrbracket(X_\lightning) = X_\lightning$ by induction over the structure of $P$:

*Case $P = \texttt{skip}$:*

$$\llbracket P \rrbracket(X_\lightning) = X_\lightning$$

*Case $P = x_i \coloneqq E$:*

$$\begin{aligned}\llbracket P \rrbracket(X_\lightning) &= X_\lightning + \sum_{\sigma \in \mathbb{N}^k} 0 \cdot X_1^{\sigma_1} \cdots X_i^{E(\sigma)} \cdots X_k^{\sigma_k}\\ &= X_\lightning\end{aligned}$$

*Case $P = \texttt{observe}\ (\,B\,)$:*

$$\begin{aligned}\llbracket P \rrbracket(X_\lightning) &= \left(1 + \left|\langle X_\lightning \rangle_{\neg B}\right|\right) X_\lightning + \langle X_\lightning \rangle_B\\ &= X_\lightning\end{aligned}$$

*Case $P = \{\,P_1\,\}\ [\,p\,]\ \{\,P_2\,\}$:*

$$\begin{aligned}\llbracket P \rrbracket(X_\lightning) &= p \cdot \llbracket P_1 \rrbracket(X_\lightning) + (1-p) \cdot \llbracket P_2 \rrbracket(X_\lightning)\\ &= p \cdot X_\lightning + (1-p) \cdot X_\lightning && \text{(I.H. on } P_1 \text{ and } P_2)\\ &= X_\lightning\end{aligned}$$

*Case $P = \texttt{if}\ (\,B\,)\ \{\,P_1\,\}\ \texttt{else}\ \{\,P_2\,\}$:*

$$\begin{aligned}\llbracket P \rrbracket(F) &= X_\lightning + \llbracket P_1 \rrbracket(\langle X_\lightning \rangle_B) + \llbracket P_2 \rrbracket(\langle X_\lightning \rangle_{\neg B})\\ &= X_\lightning + \llbracket P_1 \rrbracket(0) + \llbracket P_2 \rrbracket(0)\\ &= X_\lightning\end{aligned}$$

*Case $P = P_1 \fatsemi P_2$:*

$$\begin{aligned}\llbracket P \rrbracket(X_\lightning) &= \llbracket P_2 \rrbracket(\llbracket P_1 \rrbracket(X_\lightning))\\ &= \llbracket P_2 \rrbracket(X_\lightning) && \text{(I.H. on } P_1)\\ &= X_\lightning && \text{(I.H. on } P_2)\end{aligned}$$

*Case $P = \texttt{while}\,(\,B\,)\,\{\,P_1\,\}$:*
We show that $\forall n \in \mathbb{N} : \Phi^{n+1}_{B,P_1}(\bot)(X_{\lightning}) = X_{\lightning}$:

$$\begin{aligned}
\Phi^{n+1}_{B,P_1}(\bot)(X_{\lightning}) &= \Phi_{B,P_1}(\Phi^{n}_{B,P_1}(\bot))(X_{\lightning}) \\
&= X_{\lightning} + \langle X_{\lightning}\rangle_{\neg B} + \Phi^{n}_{B,P_1}(\bot)(\llbracket P_1\rrbracket(\langle X_{\lightning}\rangle_{B})) && (\text{def. } \Phi_{B,P_1}) \\
&= X_{\lightning} + \Phi^{n}_{B,P_1}(\bot)(0) \\
&= X_{\lightning} && (\Phi^{n}_{B,P_1}(\bot)(0) \leq \llbracket \texttt{while}\,(\,B\,)\,\{\,P_1\,\}\rrbracket(0) = 0)
\end{aligned}$$

From this, it follows:

$$\begin{aligned}
\llbracket P\rrbracket(X_{\lightning}) &= \sup\left\{\Phi^{n}_{B,P_1}(\bot)(X_{\lightning}) \mid n \in \mathbb{N}\right\} \\
&= \sup\left\{0, X_{\lightning}\right\} && (\forall n \in \mathbb{N} : \Phi^{n+1}_{B,P_1}(\bot)(X_{\lightning}) = X_{\lightning}) \\
&= X_{\lightning}
\end{aligned}$$

□

**Lemma 42 (Alternative Representation).**

$$\llbracket \texttt{while}\,(\,B\,)\,\{\,P\,\}\rrbracket(G) = \sum_{i=0}^{\infty}\left([\,\lightning\,]_{\varphi^{i}_{B,P}(G)}X_{\lightning} + \langle\varphi^{i}_{B,P}(G)\rangle_{\neg B}\right)$$
$$\textit{where} \quad \varphi_{B,P}(G) \coloneqq \llbracket P\rrbracket(\langle G\rangle_{B}).$$

Proof. First, we show by induction that for all $n \in \mathbb{N}$:

$$\Phi^{n}_{B,P}(\bot)(G) = \sum_{i=0}^{n-1}\left([\,\lightning\,]_{\varphi^{i}_{B,P}(F)}X_{\lightning} + \langle\varphi^{i}_{B,P}(F)\rangle_{\neg B}\right).$$

*Base case: $n = 0$.*

$$\Phi^{0}_{B,P}(\bot)(G) = 0 = \sum_{i=0}^{-1}\left([\,\lightning\,]_{\varphi^{i}_{B,P}(G)}X_{\lightning} + \langle\varphi^{i}_{B,P}(G)\rangle_{\neg B}\right)$$

*Induction step:*

$$\begin{aligned}
\Phi^{n+1}_{B,P}(\bot)(G) &= \Phi_{B,P}(\Phi^{n}_{B,P}(\bot))(G) \\
&= [\,\lightning\,]_{G}X_{\lightning} + \langle G\rangle_{\neg B} + \Phi^{n}_{B,P}(\bot)(\llbracket P\rrbracket(\langle G\rangle_{B})) && (\text{Def. } \Phi_{B,P}) \\
&= [\,\lightning\,]_{G}X_{\lightning} + \langle G\rangle_{\neg B} + \Phi^{n}_{B,P}(\bot)(\varphi_{B,P}(G)) && (\text{Def. } \varphi_{B,P}) \\
&= [\,\lightning\,]_{G}X_{\lightning} + \langle G\rangle_{\neg B} \\
&\quad + \sum_{i=0}^{n-1}\left([\,\lightning\,]_{\varphi^{i}_{B,P}(\varphi_{B,P}(G))}X_{\lightning} + \langle\varphi^{i}_{B,P}(\varphi_{B,P}(G))\rangle_{\neg B}\right) && (\text{I.H.}) \\
&= [\,\lightning\,]_{\varphi^{0}_{B,P}(G)}X_{\lightning} + \langle\varphi^{0}_{B,P}(G)\rangle_{\neg B} \\
&\quad + \sum_{i=1}^{n}[\,\lightning\,]_{\varphi^{i}_{B,P}(G)}X_{\lightning} + \langle\varphi^{i}_{B,P}(G)\rangle_{\neg B} && (\varphi^{0}_{B,P}(G) = G, \text{ index shift}) \\
&= \sum_{i=0}^{n}[\,\lightning\,]_{\varphi^{i}_{B,P}(G)}X_{\lightning} + \langle\varphi^{i}_{B,P}(G)\rangle_{\neg B}
\end{aligned}$$

From this, it follows:

$$\begin{aligned}
[\![\texttt{while}\,(\,B\,)\,\{\,P\,\}]\!](G) &= \left(\sup_{n\in\mathbb{N}} \Phi^{n}_{B,P}(\bot)\right)(G) \\
&= \sup\left\{\Phi^{n}_{B,P}(\bot)(G) \mid n \in \mathbb{N}\right\} \\
&= \sup_{n\in\mathbb{N}} \left\{ \sum_{i=0}^{n-1} [\lightning]_{\varphi^{i}_{B,P}(G)} X_{\lightning} + \langle \varphi^{i}_{B,P}(G)\rangle_{\neg B} \right\} \\
&= \sum_{i=0}^{\infty} [\lightning]_{\varphi^{i}_{B,P}(G)} X_{\lightning} + \langle \varphi^{i}_{B,P}(G)\rangle_{\neg B}
\end{aligned}$$

□

**Lemma 43 (Infinite Applications of Linearity).** *Let the linear and continuous function* $\psi\colon \mathsf{eFPS} \longrightarrow \mathsf{eFPS}$*. Further, let* $\alpha_i \in \mathbb{R}^{\infty}_{\geq 0}$ *and* $F_i \in \mathsf{eFPS}$ *for all* $i \in \mathbb{N}$*. Then,*

$$\psi\left(\sum_{i\in\mathbb{N}} \alpha_i \cdot F_i\right) = \sum_{i\in\mathbb{N}} \alpha_i \cdot \psi(F_i).$$

Proof.

$$\begin{aligned}
\psi\left(\sum_{i\in\mathbb{N}} \alpha_i \cdot F_i\right) &= \psi\left(\sup\left\{\sum_{i=0}^{n} \alpha_i \cdot F_i \mid n \in \mathbb{N}\right\}\right) \\
&= \sup\left\{\psi\left(\sum_{i=0}^{n} \alpha_i \cdot F_i\right) \mid n \in \mathbb{N}\right\} && \text{(Cont. of } \psi\text{)} \\
&= \sup\left\{\sum_{i=0}^{n} \alpha_i \cdot \psi(F_i) \mid n \in \mathbb{N}\right\} && \text{(finitely many applications of linearity)} \\
&= \sum_{i\in\mathbb{N}} \alpha_i \cdot \psi(F_i)
\end{aligned}$$

□

## B.1 Coincidence to Operational Semantics

We refer to the operational semantics for cpGCL programs described in [Olmedo et al. 2018]. We show that the Markov chain $\mathcal{R}_\sigma[\![P]\!]$ precisely reflects the unconditioned PGF semantics $[\![P]\!](\mathbf{X}^\sigma)$ for any $p \in$ cpGCL with initial state $\sigma \in \mathbb{N}^k$. Lemma 45 shows that the probabilities $\Pr^{\mathcal{R}_\sigma[\![P]\!]}(\Diamond\langle\downarrow,\sigma'\rangle)$ for all $\sigma'$ and $\Pr^{\mathcal{R}_\sigma[\![P]\!]}(\Diamond\lightning)$ arising from the Markov chain correspond to the coefficients of $[\![P]\!](\mathbf{X}^\sigma)$. It is further shown that modifying these probabilities to the conditional probabilities $\Pr^{\mathcal{R}_\sigma[\![P]\!]}(\Diamond\langle\downarrow,\sigma'\rangle \mid \neg\Diamond\lightning)$ has the same effect as applying the normalization function *norm*, thus concluding that the two semantics coincide (cf. Theorem 46).

**Definition 44 (Markov Chain Semantics of cpGCL).** *For any* cpGCL *program* $P$ *and any starting state* $\sigma \in \mathbb{N}^k$*, the* operational Markov chain *is*

$$\mathcal{R}_\sigma[\![P]\!] \;\triangleq\; (\mathcal{S}, \langle P,\sigma\rangle, \mathcal{P}),$$

*where:*

- $\langle P,\sigma\rangle$ *is the starting state*
- *the set of states* $\mathcal{S}$ *is the smallest set such that:*
  - $\mathcal{S}$ *contains the starting state* $\langle P,\sigma\rangle$

  - *if $s \in \mathcal{S}$ and $s$ has an outgoing transition to $s'$ according to Fig. 5, then $s' \in \mathcal{S}$*
- *$\mathcal{P} \colon \mathcal{S} \times \mathcal{S} \to [0,1]$ is the transition matrix with*
  - *$\mathcal{P}(s,s') = p$ if $s \xrightarrow{p} s'$ can be derived according to Fig. 5*
  - *$\mathcal{P}(s,s') = 0$ otherwise*

$$\text{(skip)}\ \frac{}{\langle \texttt{skip}, \sigma\rangle \longrightarrow \langle \downarrow, \sigma\rangle} \qquad \text{(asgn)}\ \frac{}{\langle x := E, \sigma\rangle \longrightarrow \langle \downarrow, \sigma[x \leftarrow E(\sigma)]\rangle}$$

$$\text{(obs-t)}\ \frac{\sigma \models B}{\langle \texttt{observe}\ (B), \sigma\rangle \longrightarrow \langle \downarrow, \sigma\rangle} \qquad \text{(obs-f)}\ \frac{\sigma \not\models B}{\langle \texttt{observe}\ (B), \sigma\rangle \longrightarrow \langle \lightning \rangle}$$

$$\text{(seq-1)}\ \frac{}{\langle \downarrow \fatsemi Q, \sigma\rangle \longrightarrow \langle Q, \sigma\rangle} \qquad \text{(seq-2)}\ \frac{\langle P, \sigma\rangle \longrightarrow \langle \lightning \rangle}{\langle P \fatsemi Q, \sigma\rangle \longrightarrow \langle \lightning \rangle}$$

$$\text{(seq-3)}\ \frac{\langle P, \sigma\rangle \xrightarrow{p} \langle P', \sigma'\rangle}{\langle P \fatsemi Q, \sigma\rangle \xrightarrow{p} \langle P' \fatsemi Q, \sigma'\rangle}$$

$$\text{(choice-l)}\ \frac{}{\langle \{P\}\ [p]\ \{Q\}, \sigma\rangle \xrightarrow{p} \langle P, \sigma\rangle} \qquad \text{(choice-r)}\ \frac{}{\langle \{P\}\ [p]\ \{Q\}, \sigma\rangle \xrightarrow{1-p} \langle Q, \sigma\rangle}$$

$$\text{(if-t)}\ \frac{\sigma \models B}{\langle \texttt{if}\ (B)\ \{P\}\ \texttt{else}\ \{Q\}, \sigma\rangle \longrightarrow \langle P, \sigma\rangle} \qquad \text{(if-f)}\ \frac{\sigma \not\models B}{\langle \texttt{if}\ (B)\ \{P\}\ \texttt{else}\ \{Q\}, \sigma\rangle \longrightarrow \langle Q, \sigma\rangle}$$

$$\text{(while-t)}\ \frac{\sigma \models B}{\langle \texttt{while}\,(B)\,\{P\}, \sigma\rangle \longrightarrow \langle P \fatsemi \texttt{while}\,(B)\,\{P\}, \sigma\rangle} \qquad \text{(while-f)}\ \frac{\sigma \not\models B}{\langle \texttt{while}\,(B)\,\{P\}, \sigma\rangle \longrightarrow \langle \downarrow, \sigma\rangle}$$

$$\text{(terminal)}\ \frac{}{\langle \downarrow, \sigma\rangle \longrightarrow \langle \mathit{sink}\rangle} \qquad \text{(undesired)}\ \frac{}{\langle \lightning \rangle \longrightarrow \langle \mathit{sink}\rangle} \qquad \text{(sink)}\ \frac{}{\langle \mathit{sink}\rangle \longrightarrow \langle \mathit{sink}\rangle}$$

Fig. 5. Construction rules for the operational Markov chain. $\sigma[x \leftarrow E(\sigma)]$ denotes the program state $\sigma$ with the value of $x$ replaced by $E(\sigma)$. Whenever a transition has no annotated weight above its arrow, it has a weight of 1.

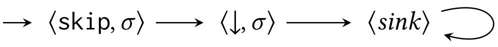
**Lemma 45.** *For every $P \in$ cpGCL and every two $\sigma, \sigma' \in \mathbb{N}^k$*

*(1)* $\Pr^{\mathcal{R}_\sigma[\![P]\!]}(\Diamond\langle\downarrow, \sigma'\rangle) \ = \ [\sigma']_{[\![P]\!](\mathbf{X}^\sigma)}$

*(2)* $\Pr^{\mathcal{R}_\sigma[\![P]\!]}(\Diamond \lightning) \ = \ [\lightning]_{[\![P]\!](\mathbf{X}^\sigma)}$

Proof. We prove the statements (1) and (2) simultaneously by structural induction over a cpGCL program $P$.

**Case** $P = \texttt{skip}$: In this case, the Markov chain $\mathcal{R}_\sigma[\![P]\!]$ looks as follows: Its PGF semantics yields:

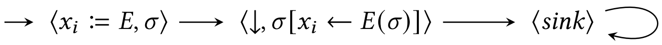


$$[\![P]\!](\mathbf{X}^\sigma) \ = \ \mathbf{X}^\sigma$$

Thus:

$$\Pr^{\mathcal{R}_\sigma[\![P]\!]}(\Diamond\langle\downarrow, \sigma'\rangle) = \begin{cases} 1, & \text{if } \sigma' = \sigma \\ 0, & \text{else} \end{cases} \ = \ [\sigma']_{[\![P]\!](\mathbf{X}^\sigma)}$$

$$\Pr^{\mathcal{R}_\sigma[\![P]\!]}(\Diamond \lightning) = 0 \ = \ [\lightning]_{[\![P]\!](\mathbf{X}^\sigma)}$$


$\langle x_i \coloneqq E, \sigma\rangle$
$\langle {\downarrow}, \sigma[x_i \leftarrow E(\sigma)]\rangle$
$\langle sink\rangle$

**Case** $P = x_i \coloneqq E$:

Its PGF semantics yields:

$$\llbracket P\rrbracket(\mathbf{X}^{\sigma}) = X_1^{\sigma_1}\cdots X_i^{E(\sigma)}\cdots X_k^{\sigma_k}$$

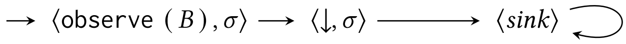
Thus:

$$\begin{aligned}\Pr^{\mathcal{R}_\sigma\llbracket P\rrbracket}(\Diamond\langle{\downarrow},\sigma'\rangle) &= \begin{cases}1, & \text{if } \sigma' = \sigma[x_i \leftarrow E(\sigma)]\\ 0, & \text{else}\end{cases}\\ &= [\sigma']_{\llbracket P\rrbracket(\mathbf{X}^{\sigma})}\end{aligned}$$

$$\Pr^{\mathcal{R}_\sigma\llbracket P\rrbracket}(\Diamond\lightning) = 0 \;=\; [\lightning]_{\llbracket P\rrbracket(\mathbf{X}^{\sigma})}$$

**Case** $P = \texttt{observe}\;(B)$: We do a case distinction whether $\sigma \models B$.

*Observe passed*:


$\langle \texttt{observe}\;(B), \sigma\rangle$
$\langle {\downarrow}, \sigma\rangle$
$\langle sink\rangle$

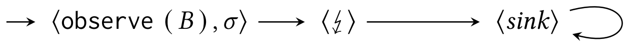
The PGF semantics yields:

$$\llbracket P\rrbracket(\mathbf{X}^{\sigma}) = \langle\mathbf{X}^{\sigma}\rangle_B + (|\langle\mathbf{X}^{\sigma}\rangle_{\neg B}| + [\lightning]_{\mathbf{X}^{\sigma}})\,X_{\lightning} \;= \mathbf{X}^{\sigma}$$

Thus:

$$\begin{aligned}\Pr^{\mathcal{R}_\sigma\llbracket P\rrbracket}(\Diamond\langle{\downarrow},\sigma'\rangle) = \begin{cases}1, & \text{if } \sigma' = \sigma\\ 0, & \text{else}\end{cases} &= [\sigma']_{\llbracket P\rrbracket(\mathbf{X}^{\sigma})}\\ \Pr^{\mathcal{R}_\sigma\llbracket P\rrbracket}(\Diamond\lightning) = 0 &= [\lightning]_{\llbracket P\rrbracket(\mathbf{X}^{\sigma})}\end{aligned}$$

*Observe failed*:


$\langle \texttt{observe}\;(B), \sigma\rangle$
$\langle \lightning\rangle$
$\langle sink\rangle$

The PGF semantics yields:

$$\llbracket P\rrbracket(\mathbf{X}^{\sigma}) = \langle\mathbf{X}^{\sigma}\rangle_B + (|\langle\mathbf{X}^{\sigma}\rangle_{\neg B}| + [\lightning]_{\mathbf{X}^{\sigma}})\,X_{\lightning} \;= X_{\lightning}$$

Thus:

$$\begin{aligned}\Pr^{\mathcal{R}_\sigma\llbracket P\rrbracket}(\Diamond\langle{\downarrow},\sigma'\rangle) = 0 &= [\sigma']_{\llbracket P\rrbracket(\mathbf{X}^{\sigma})}\\ \Pr^{\mathcal{R}_\sigma\llbracket P\rrbracket}(\Diamond\lightning) = 1 &= [\lightning]_{\llbracket P\rrbracket(\mathbf{X}^{\sigma})}\end{aligned}$$

**Case** $P = \{\,P_1\,\}\;[\,p\,]\;\{\,P_2\,\}$:

$$\begin{tikzcd}
 & & \langle P_1, \sigma \rangle \arrow[r, squiggly] & \cdots \\
{} \arrow[r] & \langle \{ P_1 \} \; [ p ] \; \{ P_2 \}, \sigma \rangle \arrow[ur, "p"] \arrow[dr, "1-p"'] & & \\
 & & \langle P_2, \sigma \rangle \arrow[r, squiggly] & \cdots
\end{tikzcd}$$

The PGF semantics yields:

$$[\![P]\!](\mathbf{X}^\sigma) = p \cdot [\![P_1]\!](\mathbf{X}^\sigma) + (1-p) \cdot [\![P_2]\!](\mathbf{X}^\sigma)$$

Thus:

$$\begin{aligned}
\Pr^{\mathcal{R}_\sigma[\![P]\!]}(\Diamond\langle\downarrow, \sigma'\rangle) &= p \cdot \Pr^{\mathcal{R}_\sigma[\![P_1]\!]}(\Diamond\langle\downarrow, \sigma'\rangle) \\
&\quad + (1-p) \cdot \Pr^{\mathcal{R}_\sigma[\![P_2]\!]}(\Diamond\langle\downarrow, \sigma'\rangle) \\
&= p \cdot [\sigma']_{[\![P_1]\!](\mathbf{X}^\sigma)} + (1-p) \cdot [\sigma']_{[\![P_2]\!](\mathbf{X}^\sigma)} && \text{(by I.H.)} \\
&= [\sigma']_{[\![P]\!](\mathbf{X}^\sigma)}
\end{aligned}$$

$$\begin{aligned}
\Pr^{\mathcal{R}_\sigma[\![P]\!]}(\Diamond \lightning) &= p \cdot \Pr^{\mathcal{R}_\sigma[\![P_1]\!]}(\Diamond \lightning) + (1-p) \cdot \Pr^{\mathcal{R}_\sigma[\![P_2]\!]}(\Diamond \lightning) \\
&= p \cdot [\lightning]_{[\![P_1]\!](\mathbf{X}^\sigma)} + (1-p) \cdot [\lightning]_{[\![P_2]\!](\mathbf{X}^\sigma)} && \text{(by I.H.)} \\
&= [\lightning]_{[\![P]\!](\mathbf{X}^\sigma)}
\end{aligned}$$

**Case** $P = \texttt{if}\ (B)\ \{P_1\}\ \texttt{else}\ \{P_2\}$: We do a case distinction on $\sigma \models B$.

*Condition is satisfied*:

$$\begin{tikzcd}
{} \arrow[r] & \langle \texttt{if}\ (B)\ \{ P_1 \}\ \texttt{else}\ \{ P_2 \}, \sigma \rangle \arrow[r] & \langle P_1, \sigma \rangle \arrow[r, squiggly] & \cdots
\end{tikzcd}$$

The PGF semantics yields:

$$[\![P]\!](\mathbf{X}^\sigma) = [\![P_1]\!](\langle \mathbf{X}^\sigma \rangle_B) + [\![P_2]\!](\langle \mathbf{X}^\sigma \rangle_{\neg B}) + [\lightning]_{\mathbf{X}^\sigma} X_\lightning \;=\; [\![P_1]\!](\mathbf{X}^\sigma)$$

Thus:

$$\begin{aligned}
\Pr^{\mathcal{R}_\sigma[\![P]\!]}(\Diamond\langle\downarrow, \sigma'\rangle) &= \Pr^{\mathcal{R}_\sigma[\![P_1]\!]}(\Diamond\langle\downarrow, \sigma'\rangle) \\
&= [\sigma']_{[\![P_1]\!](\mathbf{X}^\sigma)} && \text{(by I.H.)}
\end{aligned}$$

$$\begin{aligned}
\Pr^{\mathcal{R}_\sigma[\![P]\!]}(\Diamond \lightning) &= [\lightning]_{[\![P_1]\!](\mathbf{X}^\sigma)} \\
&= [\lightning]_{[\![P]\!](\mathbf{X}^\sigma)}
\end{aligned}$$

*Condition not satisfied*:

$$\begin{tikzcd}
{} \arrow[r] & \langle \texttt{if}\ (B)\ \{ P_1 \}\ \texttt{else}\ \{ P_2 \}, \sigma \rangle \arrow[r] & \langle P_2, \sigma \rangle \arrow[r, squiggly] & \cdots
\end{tikzcd}$$

The PGF semantics yields:

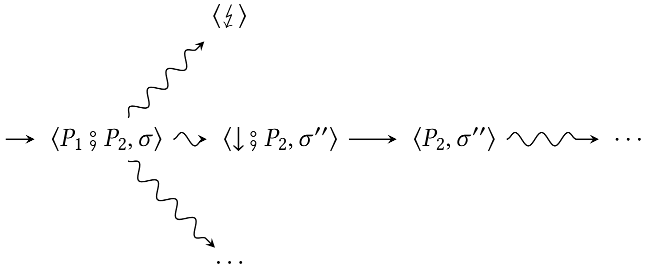
$$\llbracket P \rrbracket(\mathbf{X}^{\sigma}) = \llbracket P_1 \rrbracket(\langle \mathbf{X}^{\sigma} \rangle_{B}) + \llbracket P_2 \rrbracket(\langle \mathbf{X}^{\sigma} \rangle_{\neg B}) + [\lightning]_{\mathbf{X}^{\sigma}} X_{\lightning} \quad = \quad \llbracket P_2 \rrbracket(\mathbf{X}^{\sigma})$$

Thus:

$$\begin{aligned} \Pr^{\mathcal{R}_{\sigma}\llbracket P \rrbracket}(\Diamond\langle\downarrow, \sigma'\rangle) &= \Pr^{\mathcal{R}_{\sigma}\llbracket P_2 \rrbracket}(\Diamond\langle\downarrow, \sigma'\rangle) \\ &= [\sigma']_{\llbracket P_2 \rrbracket(\mathbf{X}^{\sigma})} && \text{(by I.H.)} \end{aligned}$$

$$\begin{aligned} \Pr^{\mathcal{R}_{\sigma}\llbracket P \rrbracket}(\Diamond \lightning) &= [\lightning]_{\llbracket P_2 \rrbracket(\mathbf{X}^{\sigma})} \\ &= [\lightning]_{\llbracket P \rrbracket(\mathbf{X}^{\sigma})} \end{aligned}$$

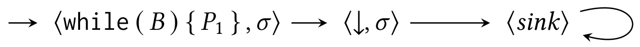
**Case** $P = P_1 \fatsemi P_2$:



$\langle \lightning \rangle$
$\langle P_1 \fatsemi P_2, \sigma \rangle$
$\langle \downarrow \fatsemi P_2, \sigma'' \rangle$
$\langle P_2, \sigma'' \rangle$
$\cdots$
$\cdots$

**Case** $P = \texttt{while}\,(\,B\,)\,\{\,P_1\,\}$:

*Condition not fulfilled* ($\sigma \not\models B$):



$\langle \texttt{while}\,(\,B\,)\,\{\,P_1\,\}, \sigma \rangle$
$\langle \downarrow, \sigma \rangle$
$\langle sink \rangle$

For the PGF semantics, consider the following, for all $n \in \mathbb{N}$:

$$\begin{aligned} \Phi^{n}_{B,P_1}(\bot)(\mathbf{X}^{\sigma}) &= [\lightning]_{\mathbf{X}^{\sigma}} + \langle \mathbf{X}^{\sigma} \rangle_{\neg B} + \Phi^{n}_{B,P_1}(\bot)(\llbracket P_1 \rrbracket(\langle \mathbf{X}^{\sigma} \rangle_{B})) \\ &= [\lightning]_{\mathbf{X}^{\sigma}} + \Phi^{n}_{B,P_1}(\bot)(\llbracket P_1 \rrbracket(\langle \mathbf{X}^{\sigma} \rangle_{B})) && (\sigma \not\models B) \\ &= 0 + \mathbf{X}^{\sigma} + 0 \\ \llbracket P \rrbracket(\mathbf{X}^{\sigma}) &= \operatorname{lfp} \Phi_{B,P_1}(\mathbf{X}^{\sigma}) \\ &= \sup_{n \in \mathbb{N}} \left\{ \Phi^{n}_{B,P_1}(\bot)(\mathbf{X}^{\sigma}) \mid n \in \mathbb{N} \right\} \\ &= \mathbf{X}^{\sigma} \end{aligned}$$

Thus:

$$\begin{aligned} \Pr^{\mathcal{R}_{\sigma}\llbracket P \rrbracket}(\Diamond\langle\downarrow, \sigma'\rangle) &= \begin{cases} 1, & \text{if } \sigma' = \sigma \\ 0, & \text{otherwise} \end{cases} \quad = \quad [\sigma']_{\llbracket P \rrbracket(\mathbf{X}^{\sigma})} \\ \Pr^{\mathcal{R}_{\sigma}\llbracket P \rrbracket}(\Diamond \lightning) &= 0 \quad = [\lightning]_{\llbracket P \rrbracket(\mathbf{X}^{\sigma})} \end{aligned}$$

*Condition is satisfied* ($\sigma \models B$):

At least one loop iteration is performed. In order for the program to terminate in some state $\sigma'$, some (non-zero) number of loop iterations must be performed. The termination probability can therefore be partitioned into the following infinite sum of probabilities:

$$\Pr^{\mathcal{R}_\sigma\llbracket P\rrbracket}(\Diamond\langle\downarrow,\sigma'\rangle) = \sum_{n=1}^{\infty} \Pr^{\mathcal{R}_\sigma\llbracket P\rrbracket}(\Diamond_{=n}\langle\downarrow,\sigma'\rangle)$$

$$\Pr^{\mathcal{R}_\sigma\llbracket P\rrbracket}(\Diamond\lightning) = \sum_{n=1}^{\infty} \Pr^{\mathcal{R}_\sigma\llbracket P\rrbracket}(\Diamond_{=n}\langle\lightning\rangle)\,,$$

where $\Pr^{\mathcal{R}_\sigma\llbracket P\rrbracket}(\Diamond_{=n}\langle\downarrow,\sigma'\rangle)$ denotes the probability to reach state $\langle\downarrow,\sigma'\rangle$ after *exactly* $n$ loop iterations and $\Pr^{\mathcal{R}_\sigma\llbracket P\rrbracket}(\Diamond_{=n}\langle\lightning\rangle)$ denotes the probability to reach state $\langle\lightning\rangle$ in exactly $n$ loop iterations.

By Lemma 42, the PGF semantics can be represented as follows:

$$\llbracket P\rrbracket(\mathbf{X}^{\sigma}) = \sum_{n=0}^{\infty}\left([\lightning]_{\varphi^{n}_{B,P_1}(\mathbf{X}^{\sigma})}X_{\lightning} + \langle\varphi^{n}_{B,P_1}(\mathbf{X}^{\sigma})\rangle_{\neg B}\right), \quad \text{where}$$

$$\varphi_{B,P_1}(F) = \llbracket P_1\rrbracket(\langle F\rangle_B).$$

By the assumption that $\sigma \models B$, the 0-th term of this series must be 0, and thus:

$$\llbracket P\rrbracket(\mathbf{X}^{\sigma}) = \sum_{n=1}^{\infty}\left([\lightning]_{\varphi^{n}_{B,P_1}(\mathbf{X}^{\sigma})}X_{\lightning} + \langle\varphi^{n}_{B,P_1}(\mathbf{X}^{\sigma})\rangle_{\neg B}\right)$$

We can therefore restate the initial claims of Lemma 45 as the following (stricter) conditions:

1. For all $n \in \mathbb{N}_{>0}$:
$$\Pr^{\mathcal{R}_\sigma\llbracket P\rrbracket}(\Diamond_{=n}\langle\downarrow,\sigma'\rangle) = [\sigma']_{\langle\varphi^{n}_{B,P_1}(\mathbf{X}^{\sigma})\rangle_{\neg B}}$$
2. For all $n \in \mathbb{N}_{>0}$:
$$\Pr^{\mathcal{R}_\sigma\llbracket P\rrbracket}(\Diamond_{=n}\langle\lightning\rangle) = [\lightning]_{\varphi^{n}_{B,P_1}(\mathbf{X}^{\sigma})}$$

For both parts, we make use of the following observation, which follows from the linearity of $\varphi$ and the assumption $\sigma \models B$:

$$\begin{aligned}
\varphi^{n+1}_{B,P_1}(\mathbf{X}^{\sigma}) &= \varphi^{n}_{B,P_1}\left(\varphi_{B,P_1}(\mathbf{X}^{\sigma})\right)\\
&= \varphi^{n}_{B,P_1}\left(\sum_{\sigma''\in\mathbb{N}^k}[\sigma'']_{\llbracket P_1\rrbracket(\mathbf{X}^{\sigma})}\mathbf{X}^{\sigma''}\right)\\
&= \sum_{\sigma''\in\mathbb{N}^k}[\sigma'']_{\llbracket P_1\rrbracket(\mathbf{X}^{\sigma})}\cdot\varphi^{n}_{B,P_1}(\mathbf{X}^{\sigma''})
\end{aligned} \tag{9}$$

1. First, note that a loop can never terminate in $\sigma'$ if $\sigma' \models B$. Accordingly, the construction rules of the Markov chain semantics (cf. Figure 5) contain the rule (while-f) as the only way of reaching a terminating state from a loop, which is only applicable if $\sigma' \not\models B$. We therefore have (for all $n \in \mathbb{N}^{k}_{>0}$):
$$\Pr^{\mathcal{R}_\sigma\llbracket P\rrbracket}(\Diamond_{=n}\langle\downarrow,\sigma'\rangle) = [\sigma']_{\langle\varphi^{n}_{B,P_1}(\mathbf{X}^{\sigma})\rangle_{\neg B}} = 0$$
We show the case $\sigma' \not\models B$ by induction:

*Base case:* $n = 1$.

$$\begin{aligned}
&\Pr^{\mathcal{R}_\sigma[\![P]\!]}(\diamond_{=1}\langle \downarrow, \sigma'\rangle)\\
&= \Pr^{\mathcal{R}_\sigma[\![P_1]\!]}(\diamond\langle \downarrow, \sigma'\rangle)\\
&= [\sigma']_{[\![P_1]\!](\mathrm{X}^\sigma)} && \text{(outer I.H.)}\\
&= [\sigma']_{\langle [\![P_1]\!](\mathrm{X}^\sigma)\rangle_{\neg B}} && (\sigma' \not\models B)\\
&= [\sigma']_{\langle [\![P_1]\!](\langle \mathrm{X}^\sigma\rangle_B)\rangle_{\neg B}} && (\sigma \models B)\\
&= [\sigma']_{\langle \varphi_{B,P_1}(\mathrm{X}^\sigma)\rangle_{\neg B}}
\end{aligned}$$

*Induction step:* In order for the loop to terminate in $n + 1$ iterations, the first execution of the loop body must terminate in some state $\sigma''$, from which the loop then terminates in $n$ iterations, i.e.,

$$\begin{aligned}
&\Pr^{\mathcal{R}_\sigma[\![P]\!]}(\diamond_{=n+1}\langle \downarrow, \sigma'\rangle)\\
&= \sum_{\sigma''\in\mathbb{N}^K} \Pr^{\mathcal{R}_\sigma[\![P_1]\!]}(\diamond\langle \downarrow, \sigma''\rangle \cdot \Pr^{\mathcal{R}_{\sigma''}[\![P]\!]}(\diamond_{=n}\langle \downarrow, \sigma'\rangle)\\
&= \sum_{\sigma''\in\mathbb{N}^k} [\sigma'']_{[\![P_1]\!](\mathrm{X}^\sigma)} \cdot \Pr^{\mathcal{R}_{\sigma''}[\![P]\!]}(\diamond_{=n}\langle \downarrow, \sigma'\rangle) && \text{(outer I.H.)}\\
&= \sum_{\sigma''\models B} [\sigma'']_{[\![P_1]\!](\mathrm{X}^\sigma)} \cdot \Pr^{\mathcal{R}_{\sigma''}[\![P]\!]}(\diamond_{=n}\langle \downarrow, \sigma'\rangle) && (0 \text{ if } \sigma'' \not\models B)\\
&= \sum_{\sigma''\models B} [\sigma'']_{[\![P_1]\!](\mathrm{X}^\sigma)} \cdot [\sigma']_{\langle \varphi^n_{B,P_1}(\mathrm{X}^{\sigma''})\rangle_{\neg B}} && \text{(inner I.H.)}\\
&= \sum_{\sigma''\in\mathbb{N}^k} [\sigma'']_{[\![P_1]\!](\mathrm{X}^\sigma)} \cdot [\sigma']_{\langle \varphi^n_{B,P_1}(\mathrm{X}^{\sigma''})\rangle_{\neg B}} && (0 \text{ if } \sigma'' \not\models B)\\
&= \sum_{\sigma''\in\mathbb{N}^k} [\sigma']_{\langle [\sigma'']_{[\![P_1]\!](\mathrm{X}^\sigma)} \cdot \varphi^n_{B,P_1}(\mathrm{X}^{\sigma''})\rangle_{\neg B}} && (\text{Lin. of } [\sigma'] \text{ and } \langle\cdot\rangle_B)\\
&= [\sigma']_{\langle \sum_{\sigma''\in\mathbb{N}^k} [\sigma'']_{[\![P_1]\!](\mathrm{X}^\sigma)} \cdot \varphi^n_{B,P_1}(\mathrm{X}^{\sigma''})\rangle_{\neg B}} && (\text{Lin. of } [\sigma'] \text{ and } \langle\cdot\rangle_B)\\
&= [\sigma']_{\langle \varphi^{n+1}_{B,P_1}(\mathrm{X}^\sigma)\rangle_{\neg B}} && \text{(by Eq. (9))}
\end{aligned}$$

2. By induction:

*Base case:* $n = 1$.

$$\begin{aligned}
&\Pr^{\mathcal{R}_\sigma[\![P]\!]}(\diamond_{=1}\langle \lightning\rangle)\\
&= \Pr^{\mathcal{R}_\sigma[\![P_1]\!]}(\diamond\lightning)\\
&= [\lightning]_{[\![P_1]\!](\mathrm{X}^\sigma)} && \text{(outer I.H.)}\\
&= [\lightning]_{[\![P_1]\!](\langle \mathrm{X}^\sigma\rangle_B)} && (\sigma \models B)\\
&= [\lightning]_{\varphi_{B,P_1}(\mathrm{X}^\sigma)}
\end{aligned}$$

*Induction step:* In order for the loop to reach $\langle \lightning\rangle$ in the $(n+1)$-th iteration, the first execution of the loop body must terminate in some state $\sigma''$, from where $\langle \lightning\rangle$ is then reached in the $n$-th iteration, i.e.,

$$\Pr^{\mathcal{R}_\sigma[\![P]\!]}(\diamond_{=n+1}\langle \lightning\rangle)$$

$$
\begin{aligned}
&= \sum_{\sigma'' \in \mathbb{N}^k} \Pr^{\mathcal{R}_\sigma \llbracket P_1 \rrbracket}(\Diamond\langle \downarrow, \sigma''\rangle) \cdot \Pr^{\mathcal{R}_{\sigma''} \llbracket P \rrbracket}(\Diamond_{=n}\langle \lightning \rangle) \\
&= \sum_{\sigma'' \in \mathbb{N}^k} [\sigma'']_{\llbracket P_1 \rrbracket(\mathbf{X}^\sigma)} \cdot \Pr^{\mathcal{R}_{\sigma''} \llbracket P \rrbracket}(\Diamond_{=n}\langle \lightning \rangle) && \text{(outer I.H.)} \\
&= \sum_{\sigma'' \models B} [\sigma'']_{\llbracket P_1 \rrbracket(\mathbf{X}^\sigma)} \cdot \Pr^{\mathcal{R}_{\sigma''} \llbracket P \rrbracket}(\Diamond_{=n}\langle \lightning \rangle) && (0 \text{ if } \sigma'' \not\models B) \\
&= \sum_{\sigma'' \models B} [\sigma'']_{\llbracket P_1 \rrbracket(\mathbf{X}^\sigma)} \cdot [\lightning]_{\varphi^n_{B,P_1}}(\mathbf{X}^{\sigma''}) && \text{(inner I.H.)} \\
&= \sum_{\sigma'' \in \mathbb{N}^k} [\sigma'']_{\llbracket P_1 \rrbracket(\mathbf{X}^\sigma)} \cdot [\lightning]_{\varphi^n_{B,P_1}}(\mathbf{X}^{\sigma''}) && (0 \text{ if } \sigma'' \not\models B) \\
&= \sum_{\sigma'' \in \mathbb{N}^k} [\lightning]_{[\sigma'']_{\llbracket P_1 \rrbracket(\mathbf{X}^\sigma)} \cdot \varphi^n_{B,P_1}}(\mathbf{X}^{\sigma''}) && (\text{Lin. of } [\lightning]) \\
&= [\lightning]_{\sum_{\sigma'' \in \mathbb{N}^k} [\sigma'']_{\llbracket P_1 \rrbracket(\mathbf{X}^\sigma)} \cdot \varphi^n_{B,P_1}}(\mathbf{X}^{\sigma''}) && (\text{Lin. of } [\lightning] \text{ and } \langle \cdot \rangle_B) \\
&= [\lightning]_{\varphi^{n+1}_{B,P_1}}(\mathbf{X}^\sigma) && \text{(by Eq. (9))}
\end{aligned}
$$

□

**Theorem 46 (Operational Equivalence).** *For every* cpGCL *program $p$ and every $\sigma, \sigma' \in \mathbb{N}^k$*

$$
Pr^{\mathcal{R}_\sigma} \llbracket P \rrbracket (\Diamond\langle \downarrow, \sigma' \rangle \mid \neg\Diamond\lightning) \; = \; [\sigma']_{norm(\llbracket P \rrbracket(\mathbf{X}^\sigma))} \; .
$$

*This includes the case of undefined semantics, i.e., the left-hand side is undefined if and only if the right-hand side is undefined.*

Proof.

$$
\begin{aligned}
\Pr^{\mathcal{R}_\sigma \llbracket P \rrbracket}(\Diamond\langle \downarrow, \sigma' \rangle \mid \neg\Diamond\lightning) &= \frac{\Pr^{\mathcal{R}_\sigma \llbracket P \rrbracket}(\Diamond\langle \downarrow, \sigma' \rangle \wedge \neg\Diamond\lightning)}{\Pr^{\mathcal{R}_\sigma \llbracket P \rrbracket}(\neg\Diamond\lightning)} \\
&= \frac{\Pr^{\mathcal{R}_\sigma \llbracket P \rrbracket}(\Diamond\langle \downarrow, \sigma' \rangle)}{\Pr^{\mathcal{R}_\sigma \llbracket P \rrbracket}(\neg\Diamond\lightning)} && (\text{reaching } \langle \downarrow, \sigma' \rangle \text{ implies not reaching } \langle \lightning \rangle) \\
&= \frac{\Pr^{\mathcal{R}_\sigma \llbracket P \rrbracket}(\Diamond\langle \downarrow, \sigma' \rangle)}{1 - \Pr^{\mathcal{R}_\sigma \llbracket P \rrbracket}(\Diamond\lightning)} \\
&= \frac{[\sigma']_{\llbracket P \rrbracket(\mathbf{X}^\sigma)}}{1 - [\lightning]_{\llbracket P \rrbracket(\mathbf{X}^\sigma)}} && (\text{cf. Lemma 45}) \\
&= [\sigma']_{norm(\llbracket P \rrbracket(\mathbf{X}^\sigma))}
\end{aligned}
$$

□

## C REASONING ABOUT LOOPS

**Definition 47 (Admissible eSOP-transformer).** *A function $\psi \colon$* eSOP $\to$ eSOP *is called* admissible *if*

- $\psi$ *is* continuous *on* eSOP.
- $\psi$ *is* linear *in the following sense: For all $F, G \in$* eSOP *and $p \in [0, 1]$*

$$
pF + G \in \text{eSOP} \quad \textit{implies} \quad \psi(pF + G) \; = \; p\psi(F) + \psi(G) \; .
$$

- $\psi$ *is* homogeneous w.r.t. meta indeterminates*, i.e., for all $G \in$* eSOP *and $\tau \in \mathbb{N}^l$,*

$$
\psi(G\mathbf{U}^\tau) \; = \; \psi(G)\mathbf{U}^\tau \; .
$$

- $\psi$ preserves ePGF, *i.e.,* $G \in$ ePGF *implies* $\psi(G) \in$ ePGF.

**Theorem 48** (SOP **Semantics).** *Let $P$ be a loop-free* `cReDiP` *Program. Let* $G = \sum_{\sigma \in \mathbb{N}^k} G_\sigma \mathbf{U}^\sigma \in$ eSOP. *The* eSOP *semantics of $P$ is then given by:*

$$\llbracket P \rrbracket (G) = \sum_{\sigma \in \mathbb{N}^k} \llbracket P \rrbracket (G_\sigma) \cdot \mathbf{U}^\sigma$$

Proof outline. The proof of Theorem 12 proceeds along a similar line of reasoning as in [Chen et al. 2022a]. First of all, we use Lemma 43 and obtain:

$$\begin{aligned}
\llbracket P \rrbracket (G) &= \sum_{\sigma \in \mathbb{N}^k} \llbracket P \rrbracket (G_\sigma \mathbf{U}^\sigma) \\
&= \sum_{\sigma \in \mathbb{N}^k} \llbracket P \rrbracket \left( \sum_{\tau \in \mathbb{N}^k} [\tau]_{G_\sigma} \mathbf{X}^\tau \mathbf{U}^\sigma + [\lightning]_{G_\sigma} X_\lightning \mathbf{U}^\sigma \right) && \text{(by eFPS arithmetic)} \\
&= \sum_{\sigma \in \mathbb{N}^k} \sum_{\tau \in \mathbb{N}^k} \llbracket P \rrbracket \left( [\tau]_{G_\sigma} \mathbf{X}^\tau \mathbf{U}^\sigma + [\lightning]_{G_\sigma} X_\lightning \mathbf{U}^\sigma \right) && \text{(by Lemma 43)} \\
&= \sum_{\sigma \in \mathbb{N}^k} \sum_{\tau \in \mathbb{N}^k} \llbracket P \rrbracket \left( [\tau]_{G_\sigma} \mathbf{X}^\tau \mathbf{U}^\sigma \right) + \llbracket P \rrbracket \left( [\lightning]_{G_\sigma} X_\lightning \mathbf{U}^\sigma \right) && \text{(by Theorem 40)} \\
&= \sum_{\sigma \in \mathbb{N}^k} \sum_{\tau \in \mathbb{N}^k} [\tau]_{G_\sigma} \cdot \llbracket P \rrbracket \left( \mathbf{X}^\tau \mathbf{U}^\sigma \right) + [\lightning]_{G_\sigma} X_\lightning \mathbf{U}^\sigma && \text{(by Lemma 11, Theorem 40)}
\end{aligned}$$

Thus we need to show, that $\llbracket P \rrbracket (\mathbf{X}^\tau \mathbf{U}^\sigma)$ is homogeneous for all $\tau, \sigma \in \mathbb{N}$. All loop-free cases but `observe` coincide with `ReDiP` [Chen et al. 2022a] on the distributions where the observe violation probability is zero which is an immediate consequence of Lemma 11 and the definition in Table 3.

To complete the proof, we show that `observe ( false )` also has this property by showing that its semantics is admissible. Recall the `observe ( false )` semantics: $G[\mathbf{X}/\mathbf{1}, X_\lightning/1] \cdot X_\lightning$. Note that the `observe ( false )` semantics is entirely based on the following elementary transformations, which are admissible (by [Chen et al. 2022a]):

- Multiplication by a constant $G \in$ ePGF: $\lambda F.\ G \cdot F$
- Substitution of $X \in \mathbf{X}$ by a constant $G \in$ ePGF: $\lambda F.\ F[X/G]$.

Thus, $\llbracket$`observe ( false )`$\rrbracket\ (\mathbf{X}^\tau \mathbf{U}^\sigma) = \llbracket$`observe ( false )`$\rrbracket\ (\mathbf{X}^\tau)\ \mathbf{U}^\sigma$ holds. Another, direct argument is as follows: It is important to understand the second column of Table 3 correctly, especially for the `observe ( false )` statement. Here, $\mathbf{X}$ explicitly refers to the program variables occurring in $P$, *not* for the meta-indeterminates $\mathbf{U}$. In case we would also substitute these values by 1, we would loose all information gathered by the second order approach. Thus applying the semantics correctly, we get $\llbracket$`observe ( false )`$\rrbracket\ (\mathbf{X}^\tau \mathbf{U}^\sigma) = \mathbf{1}^\tau X_\lightning \mathbf{U}^\sigma = \llbracket$`observe ( false )`$\rrbracket\ (\mathbf{X}^\tau)\ \mathbf{U}^\sigma$. Combining that with the equation from above, we conclude

$$\begin{aligned}
\sum_{\sigma \in \mathbb{N}^k} \sum_{\tau \in \mathbb{N}^k} [\tau]_{G_\sigma} \cdot \llbracket P \rrbracket \left( \mathbf{X}^\tau \mathbf{U}^\sigma \right) + [\lightning]_{G_\sigma} X_\lightning \mathbf{U}^\sigma &= \sum_{\sigma \in \mathbb{N}^k} \sum_{\tau \in \mathbb{N}^k} [\tau]_{G_\sigma} \cdot \llbracket P \rrbracket \left( \mathbf{X}^\tau \right) \mathbf{U}^\sigma + [\lightning]_{G_\sigma} X_\lightning \mathbf{U}^\sigma \\
&= \sum_{\sigma \in \mathbb{N}^k} \sum_{\tau \in \mathbb{N}^k} \llbracket P \rrbracket \left( [\tau]_{G_\sigma} \cdot \mathbf{X}^\tau \right) \mathbf{U}^\sigma + \llbracket P \rrbracket \left( [\lightning]_{G_\sigma} X_\lightning \right) \mathbf{U}^\sigma \\
& \qquad \text{(by Theorem 40)} \\
&= \sum_{\sigma \in \mathbb{N}^k} \sum_{\tau \in \mathbb{N}^k} \left( \llbracket P \rrbracket \left( [\tau]_{G_\sigma} \cdot \mathbf{X}^\tau \right) + \llbracket P \rrbracket \left( [\lightning]_{G_\sigma} X_\lightning \right) \right) \mathbf{U}^\sigma \\
& \qquad \text{(by eFPS arithmetic)}
\end{aligned}$$

$$= \sum_{\sigma\in\mathbb{N}^k}\sum_{\tau\in\mathbb{N}^k} \left([\![P]\!]\left([\tau]_{G_\sigma}\cdot \mathbf{X}^\tau + [\lightning]_{G_\sigma} X_{\lightning}\right)\right)\mathbf{U}^\sigma \qquad \text{(Theorem 40)}$$

$$= \sum_{\sigma\in\mathbb{N}^k} [\![P]\!]\,(G_\sigma)\,\mathbf{U}^\sigma \qquad \text{(by Def. of } G_\sigma\text{)}$$

□

**Lemma 49** (eSOP **Characterization).** *Let $P_1$ and $P_2$ be loop-free* cReDiP*-programs with $Vars(P_i) \subseteq \{x_1,\ldots,x_k\}$ for $i\in\{1,2\}$. Further, consider a vector $\mathbf{U}=(U_1,\ldots,U_k)$ of meta indeterminates, and let $\hat{G}$ be the eSOP $(1-X_1U_1)^{-1}\cdots(1-X_kU_k)^{-1}\in\mathbb{R}[[\mathbf{X},\mathbf{U}]]$. Then,*

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

Proof. We observe that $\hat{G} \;=\; \sum_{\sigma\in\mathbb{N}^k}\mathbf{X}^\sigma\mathbf{U}^\sigma$. Then we have

$$\begin{aligned}
& [\![P_1]\!](\hat{G}) = [\![P_2]\!](\hat{G}) \\
\Longleftrightarrow\quad & [\![P_1]\!](\hat{G}) - [\![P_2]\!](\hat{G}) = 0 \\
\Longleftrightarrow\quad & [\![P_1]\!](\sum_{\sigma\in\mathbb{N}^k}\mathbf{X}^\sigma\mathbf{U}^\sigma) - [\![P_2]\!](\sum_{\sigma\in\mathbb{N}^k}\mathbf{X}^\sigma\mathbf{U}^\sigma) = 0 \\
\Longleftrightarrow\quad & \sum_{\sigma\in\mathbb{N}^k}[\![P_1]\!](\mathbf{X}^\sigma)\mathbf{U}^\sigma - \sum_{\sigma\in\mathbb{N}^k}[\![P_2]\!](\mathbf{X}^\sigma)\mathbf{U}^\sigma = 0 && \text{(By Theorem 12)} \\
\Longleftrightarrow\quad & \sum_{\sigma\in\mathbb{N}^k}([\![P_1]\!](\mathbf{X}^\sigma) - [\![P_2]\!](\mathbf{X}^\sigma))\mathbf{U}^\sigma = 0 && \text{(rewriting)} \\
\Longleftrightarrow\quad & \forall\sigma\in\mathbb{N}^k\colon [\![P_1]\!](\mathbf{X}^\sigma) - [\![P_2]\!](\mathbf{X}^\sigma) = 0 && \text{(By definition of the 0-FPS in } \mathbb{R}[[\mathbf{X},X_{\lightning},\mathbf{U}]]\text{)} \\
\Longleftrightarrow\quad & \forall\sigma\in\mathbb{N}^k\colon [\![P_1]\!](\mathbf{X}^\sigma) = [\![P_2]\!](\mathbf{X}^\sigma) \\
\Longleftrightarrow\quad & [\![P_1]\!] = [\![P_2]\!] && \text{(by Kozen [1981] and Lemma 11)}
\end{aligned}$$

□

## D PARAMETER SYNTHESIS

**Theorem 50 (Decidability of Parameter Synthesis).** *Let $W$ be a* `cReDiP` `while` *loop and $I_\mathbf{p}$ be a parametrized loop-free* `cReDiP` *program. It is decidable whether there exist parameter values $\rho$ such that the instantiated template $I_\rho$ is an invariant, i.e.,*

$$\exists\mathbf{p}\in\mathbb{R}^l.\quad [\![W]\!] = [\![I_\mathbf{p}]\!]\,.$$

Proof. Let $W$ and $I_\mathbf{p}$ be given as described. Also, let $\hat{G}=(1-X_1U_1)^{-1}\cdots(1-X_kU_k)^{-1}\in\text{eSOP}$ which is a rational closed form.

$$\begin{aligned}
& \exists\mathbf{p}\in\mathbb{R}^l. && [\![I_\mathbf{p}]\!] = [\![\Phi_{B,P}(I_\mathbf{p})]\!] \\
\Leftrightarrow\ & \exists\mathbf{p}\in\mathbb{R}^l. && [\![I_\mathbf{p}]\!](\hat{G}) = [\![\Phi_{B,P}(I_\mathbf{p})]\!](\hat{G}) && \text{(Lemma 13)} \\
\Leftrightarrow\ & \exists\mathbf{p}\in\mathbb{R}^l. && \frac{F_\mathbf{p}}{H_\mathbf{p}} = \frac{\hat{F}_\mathbf{p}}{\hat{H}_\mathbf{p}} && \text{(loop-free cReDiP preserves rational functions)} \\
\Leftrightarrow\ & \exists\mathbf{p}\in\mathbb{R}^l. && F_\mathbf{p}\hat{H}_\mathbf{p} = \hat{F}_\mathbf{p}H_\mathbf{p} \\
\Leftrightarrow\ & \exists\mathbf{p}\in\mathbb{R}^l. && F_\mathbf{p}\hat{H}_\mathbf{p} - \hat{F}_\mathbf{p}H_\mathbf{p} = 0
\end{aligned}$$

In the last step, $F_\mathbf{p}\hat{H}_\mathbf{p}$ and $\hat{F}_\mathbf{p}H_\mathbf{p}$ are polynomials in $\mathbb{R}[\mathbf{p}][\mathbf{X},X_{\lightning},\mathbf{U}]$ ($\mathbf{p}$ can only occur as probabilities in $I_\mathbf{p}$). Using the results about quantifier elimination in the theory of non-linear real arithmetic (by

Cylindrical Algebraic Decomposition [Caviness and Johnson 2012]), we have a decision procedure of $O\left(2^{2^{|X|+|U|+1}}\right)$ worst-case complexity to decide whether the formula can be satisfied. □

## E BENCHMARKS AND ADDITIONAL EXAMPLES

Table 5 depicts the quantitative results for loop-free programs. The column Program lists the benchmarks. The next column ($\infty$) marks the occurrence of samplings from infinite-support distributions in the benchmark. Column $p$ indicates the presence of symbolic parameters. Finally, columns SymPy, GiNaC, symbolic, dp and genfer depict run-times in seconds for the individual backends of Prodigy, $\lambda$-PSI, and the tool genfer respectively. The timing in boldface marks the fastest variant. The acronym TO stands for time-out, i.e., did not terminate within the time limit. Entries consisting of "—" indicate the lack of support for this benchmark instance. Timings marked with * refer to results by $\lambda$-PSI which contain integral expressions. Strictly speaking, these are solution representations we like to avoid, however $\lambda$-PSI is still able to compute all moments exactly. With respect to the loop-free benchmarks, genfer is the fastest for most of the benchmarks, however it does not support parameters. $\lambda$-PSI with optimal settings and Prodigy are head to head, oftentimes yielding very similar timings. Comparing the Prodigy backends solely, GiNaC is faster by roughly two orders of magnitude.

[13] Exceeding SymPy internal limits for parsing.

[14] Reached maximum recursion limit

Table 5. Benchmarks of loop-free programs; timings are in seconds.

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

**Example 51 (The Invariant for Prog. 12).**

```
if ( $s > 0 \land f < 5$ ) {
  if ( $s \geq 10$ ) {
    if ( iid (bernoulli ($1/100$) , $s - 9$) $= 0$ ) { skip } else {observe ( false )}⨟
    $s \coloneqq 9$⨟
    $f \coloneqq 0$}
  if ( iid (bernoulli ($99/100$) , $5 - f$) $> 0$ ) {
    $f \coloneqq 0$⨟
    $s \coloneqq s - 1$⨟
    if ( $s = 8$ ) {
      { $f \coloneqq 5$ } [ $p$ ] { $f \coloneqq 0$ ⨟ $s \coloneqq s - 1$ }
    }⨟
    if ( $s = 7$ ) {
      { $f \coloneqq 5$ } [ $p$ ] { $f \coloneqq 0$ ⨟ $s \coloneqq s - 1$ }
    }⨟
    if ( $s = 6$ ) {
      { $f \coloneqq 5$ } [ $p$ ] { $f \coloneqq 0$ ⨟ $s \coloneqq s - 1$ }
    }⨟
    if ( $s = 5$ ) {
      { $f \coloneqq 5$ } [ $p$ ] { $f \coloneqq 0$ ⨟ $s \coloneqq s - 1$ }
    }⨟
    if ( $s = 4$ ) {
      { $f \coloneqq 5$ } [ $p$ ] { $f \coloneqq 0$ ⨟ $s \coloneqq s - 1$ }
    }⨟
    if ( $s = 3$ ) {
      { $f \coloneqq 5$ } [ $p$ ] { $f \coloneqq 0$ ⨟ $s \coloneqq s - 1$ }
    }⨟
    if ( $s = 2$ ) {
      { $f \coloneqq 5$ } [ $p$ ] { $f \coloneqq 0$ ⨟ $s \coloneqq s - 1$ }
    }⨟
    if ( $s = 1$ ) {
      { $f \coloneqq 5$ } [ $p$ ] { $f \coloneqq 0$ ⨟ $s \coloneqq s - 1$ }
    }
  } else {$f \coloneqq 5$}}
```

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

The inferred transmission-failure probability is

$$\frac{516019999999493340000000469819999999615440000000257299999998695400000004609999999899900000001}{51711 \cdot 10^{88}} .$$

**Example 52.** This example of two programs $I$ and $J$ shows the step-by-step computation of the invariant and the modified invariant to prove the actual equivalence.

```
I :   $/\!\!/\!\!/\; (1-XU)^{-1}(1-YV)^{-1}$
      if ( $y = 1$ ) {
          $/\!\!/\!\!/\; (1-XU)^{-1}YV$
          $x$ += iid (geom (1/2) + 1, $y$)$\fatsemi$
          $/\!\!/\!\!/\; (1-XU)^{-1}X(2-X)^{-1}YV$
          $y \coloneqq 0\fatsemi$
          $/\!\!/\!\!/\; (1-XU)^{-1}X(2-X)^{-1}V$
          observe ( $x < 3$ )
          $/\!\!/\!\!/\; (1/2X + 1/4X^2 + 1/4X_{\lightning})V$
              $+ \left((1/2X^2 + 1/2X_{\lightning})U + X_{\lightning}U^2(1-U)^{-1}\right) V$
      }
      $/\!\!/\!\!/\; (1-XU)^{-1}(1-YV)^{-1} - (1-XU)^{-1}YV$
          $+(1/2X + 1/4X^2 + 1/4X_{\lightning})V$
          $+ \left((1/2X^2 + 1/2X_{\lightning})U + X_{\lightning}U^2(1-U)^{-1}\right) V$
```

We want to show that $[\![J]\!](\hat{G})$ — where $J =$ `if` ( $y = 1$ ) $\{P \fatsemi I\}$ `else` {`skip`} — yields the same result:

```
J :   $/\!\!/\!\!/\; (1-XU)^{-1}(1-YV)^{-1}$
      if ( $y = 1$ ) {
          $/\!\!/\!\!/\; (1-XU)^{-1}YV$
          { $y \coloneqq 0$ } [ 1/2 ] { $y \coloneqq 1$ }$\fatsemi$
          $/\!\!/\!\!/\; 1/2(1-XU)^{-1}(Y+1)V$
          $x \coloneqq x + 1\fatsemi$
          $/\!\!/\!\!/\; 1/2X(1-XU)^{-1}(Y+1)V$
          observe ( $x < 3$ )$\fatsemi$
          $/\!\!/\!\!/\; \left(1/2(X + X^2U)(Y+1) + X_{\lightning}U^2(1-U)^{-1}\right) V$
          if ( $y = 1$ ) {
              $/\!\!/\!\!/\; 1/2(X + X^2U)YV$
              $x$ += iid (geom (1/2) + 1, $y$)$\fatsemi$
              $/\!\!/\!\!/\; 1/2(X^2 + X^3U)(2-X)^{-1}YV$
              $y \coloneqq 0\fatsemi$
              $/\!\!/\!\!/\; 1/2(X^2 + X^3U)(2-X)^{-1}V$
```

```
                observe ( $x < 3$ )
                /// $1/2\left(1/2X^2 + 1/2X_{\lightning} + X_{\lightning}U\right)V$
            }
            /// $\left((1/2X + 1/4X^2 + 1/4X_{\lightning}) + (1/2X^2 + 1/2X_{\lightning})U\right)V$
                $+X_{\lightning}U^2(1-U)^{-1}V$
        }
        /// $(1-XU)^{-1}(1-YV)^{-1} - (1-XU)^{-1}YV$
            $+\left((1/2X + 1/4X^2 + 1/4X_{\lightning}) + (1/2X^2 + 1/2X_{\lightning})U\right)V$
            $+X_{\lightning}U^2(1-U)^{-1}V$
```