

Exact Bayesian Inference for Loopy Probabilistic Programs

LUTZ KLINKENBERG, RWTH Aachen University, Germany

CHRISTIAN BLUMENTHAL, RWTH Aachen University, Germany

MINGSHUAI CHEN, Zhejiang University, China

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 looping behaviors*. Our method is built on an extended 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 outperforms state-of-the-art exact inference tools over benchmarks of loop-free programs.

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, and Joost-Pieter Katoen. 2024. Exact Bayesian Inference for Loopy Probabilistic Programs. *Proc. ACM Program. Lang.* 1, POPL, Article 1 (January 2024), 50 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] and probabilistic machine learning (ML) [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 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

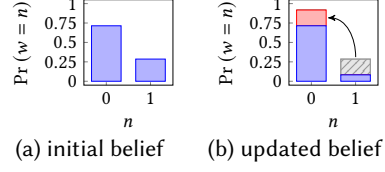
Authors' addresses: Lutz Klinkenberg, RWTH Aachen University, Aachen, Germany; Christian Blumenthal, christian.blumenthal@rwth-aachen.de, RWTH Aachen University, Aachen, Germany; Mingshuai Chen, m.chen@zju.edu.cn, Zhejiang University, Hangzhou, China; Joost-Pieter Katoen, katoen@cs.rwth-aachen.de, RWTH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

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

derive a program’s posterior distribution. In contrast to sampling-based approximate inference, inferring the *exact* distribution is of 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 *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. 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. 2022] and proving non-almost-sure termination [Klinkenberg et al. 2020] for certain programs *without conditioning*. Orthogonally, Zaiser et al. [2023] recently employed PGFs to do 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 [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.

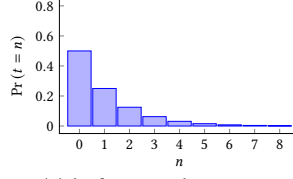
Conditioning in loop-free programs. Consider the loop-free program Prog. 1 producing an *infinite-support* distribution. It describes a telephone operator who is unaware of whether today is a weekday or weekend. The operator’s initial belief is that with probability $5/7$ it is a weekday ($w = 0$) and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

Conditioning outside loops. Prog. 2 describes an iterative algorithm that repeatedly flips a fair coin – while counting the number of trials (t) – until seeing tails ($h = 0$), and observes that this number is odd. In fact, the while-loop produces a geometric distribution in t (cf. Fig. 1a), after which the observe statement “blocks” all program runs where t is not an odd number and normalizes the

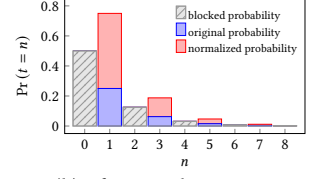
```

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

```



(a) before conditioning



(b) after conditioning

Prog. 2. The odd geometric distribution.

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

probabilities of the remaining runs (cf. Fig. 1b). Note that Prog. 2 features an unbounded looping behavior (inducing an infinite-support distribution) whose exact output distribution thus cannot be inferred by state-of-the-art inference engines, e.g., neither by (λ) PSI [Gehr et al. 2016, 2020], nor by the PGF-based approach in [Zaiser et al. 2023]. However, given a suitable invariant, our tool is able to derive the posterior distribution of Prog. 2 in an automated fashion leveraging existing computer algebra packages like SymPy [Meurer et al. 2017] and GiNAC [Bauer et al. 2002; Vollinga 2006]: The resulting output distribution for any input with $t = 0$, represented as closed-form PGF, is

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

where T and H are formal *indeterminates* corresponding to the program variables t and h , respectively. From this closed-form GF, 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.

Conditioning inside i.i.d. loops. As argued by Olmedo 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 program run that consistently visits the left branch of the probabilistic choice.

```

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

```

Prog. 3. observe inside loop.

Intuitively, the observe statement prevents the 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 [Chen et al. 2022; 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 non-termination. Zaiser et al. [2023] circumvent such semantic intricacies by syntactically imposing certainly terminating programs (due to the absence of loops and recursion). In our approach, we address these problems by specifically distinguishing non-termination behaviors from observe violations, which allows us to show that the loop in Prog. 3 is in fact equivalent to $\text{if } (h = 1) \{ \text{observe } (\text{false}) \} \text{ else } \{ \text{skip} \} \equiv \text{observe } (h \neq 1)$.

There are alternative approaches that detour the abovementioned semantic intricacies by transforming the conditioned loop into a semantically equivalent loop that has no observation inside. These approaches include (1) the *hoisting technique* [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 is the case for Prog. 3; (3) the *ad hoc solution* that simply pulls the observe statement outside of 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 (b_1 and b_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 of the loop. However, given a suitable invariant – in the form of a conditioned loop-free program that can be shown semantically 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 $b_1 = b_2 = b'_1 = b'_2 = 0$ after termination). Furthermore, our inference framework admits *parameters* in both programs and invariants, for e.g. encoding distributions with unknown probabilities such as `bernoulli(p)` with $p \in (0, 1)$; it is capable of determining possible valuations of these parameters such that the given invariant is semantically equivalent to the loop in question. The support of parameters makes it a viable approach to facilitating template-based invariant synthesis (see, e.g., [Batz et al. 2023]) and model repair (cf. [Češka et al. 2019]). See details and examples in Section 5.

```

 $n := 0$  ;
 $b_1 := 1$  ;  $b_2 := 1$  ;  $b'_1 := 1$  ;  $b'_2 := 1$  ;
while ( $\neg (b_1 = 0 \wedge b_2 = 0)$ ) {
   $b_1 := \text{bernoulli}(1/2)$  ;
   $b_2 := \text{bernoulli}(1/2)$  ;
  observe ( $b_1 = b'_1 \vee b_2 = b'_2$ ) ;
   $b'_1 := b_1$  ;
   $b'_2 := b_2$  ;
   $n := n + 1$ 
}
```

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

Contributions. The main results of this paper are the following:

- 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. 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. 2022; Klinkenberg et al. 2020] for unconditioned programs and is shown to coincide with a 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 outperforms the state-of-the-art inference tool λ -PSI over benchmarks of loop-free programs.

Paper structure. Section 2 recapitulates 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

In mathematics, *generating functions* (GF) are a wide spread tool providing answers to questions of combinatorial nature, e.g., in enumerative combinatorics [Flajolet and Sedgewick 2009; Wilf 2005] and (discrete) probability theory [Johnson et al. 2005]. For the latter, they are a useful tool to describe discrete probability distributions over the non-negative integers \mathbb{N} in a compact manner.

2.1 Univariate Generating Functions

The simplest version are *univariate* generating functions. More precisely, a univariate generating function is a possibly infinite sequence $(a_n)_{n \in \mathbb{N}}$ of numbers, labeled by a formal indeterminate X and aggregated in a formal summation $G(X) = \sum_{n=0}^{\infty} (a_n)X^n$. For instance, the sequence $1/2, 1/4, 1/8, \dots$ is encoded as

$$G(X) = \sum_{n=0}^{\infty} \frac{1}{2^{n+1}} X^n = \frac{1}{2} + \frac{1}{4}X + \frac{1}{8}X^2 + \frac{1}{16}X^3 + \frac{1}{32}X^4 + \dots, \quad (1)$$

which represents the probabilities of a geometrically distributed random variable with parameter $1/2$.

Oftentimes infinite sums are not very practical, thus we strive for *implicit* expressions in terms of a *function*, which we call closed-forms. A common way to relate an (infinite) *sequence* of numbers to an implicit encoding of a generating function uses the Taylor series expansion, i.e., one finds a function $X \mapsto G(X)$ whose Taylor series around $X = 0$ uses the numbers in the sequence as coefficients. For our concrete example, the closed-form of Equation (1) is given by $X \mapsto \frac{1}{2-X}$ for all $|X| < 2$. We emphasize that in contrast to the infinite sequence, a closed-form is –from a purely syntactical point of view– a *finite* object. Nevertheless, many important operations on infinite sequences –and their GF series representation– can be simulated by applying standard manipulations to the closed form, see Table 1 that summarizes some basic ones. In our geometric sequence for instance, taking the formal derivative $\frac{d}{dX} \frac{1}{2-X} = \frac{1}{(2-X)^2}$ and evaluating the sum at $X = 1$ yields 1 which is precisely the expected value for a random variable distributed alike.

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

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

2.2 Multivariate Generating Functions

Our aim is to compute the posterior distribution over the terminating states of probabilistic a program. Usually, programs consist of several program variables, hence a program state is defined as a tuple $(X_1, \dots, X_k) \in \mathbb{N}^k$, where $k < \infty$ is the number of program variables. Consider the program $x := \text{geom}(1/2)$, which describes a single instruction that says “sample program variable x from a geometric distribution with parameter $1/2$ ”. Its terminating states are all values (i) for $i \in \mathbb{N}^1$ and they individually occur with probability $1/2^{i+1}$. This distribution can be precisely described by Equation (1) where the exponents of X denotes the program state, and the coefficients $1/2, 1/4, 1/8, \dots$ the probabilities of reaching that state. Likewise, $\frac{1}{2-X}$ describes the *precise semantics* of the program by means of a *single fraction*.

When programs consist of multiple variables, computing closed-forms by using multidimensional Taylor series quickly becomes unhandy. One alternative approach is to define closed-forms *algebraically* via inverse elements in the ring of *Formal Power Series* (FPS). Let us recall the algebraic definition of a (commutative) *ring* $(R, +, \cdot, 0, 1)$ structure. A ring consists of a non-empty carrier set R , together with two associative and commutative binary operations: “+” (addition) and “ \cdot ” (multiplication). Additionally, we require that multiplication distributes over addition, and the existence of neutral elements 0 and 1 w.r.t. addition and multiplication, respectively. Further, every $r \in R$ has an additive inverse $-r \in R$. Multiplicative inverses $r^{-1} = 1/r$ need not always exist. For the remainder of the section, we fix $k \in \mathbb{N} = \{0, 1, \dots\}$.

Definition 1 (The Ring of FPS [Chen et al. 2022]). A k -dimensional FPS is a k -dim. array $[\cdot]_F: \mathbb{N}^k \rightarrow \mathbb{R}$. We denote FPS as formal sums as follows: Let $\mathbf{X}=(X_1, \dots, X_k)$ be an ordered vector of symbols, called *indeterminates*. The FPS F over \mathbf{X} is written as

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

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

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

The multiplication $F \cdot G$ is in fact the discrete convolution of the two sequences F and G . Sometimes, this is also called the *Cauchy product* of power series. Note that $F \cdot G$ is always well-defined because for all $\sigma \in \mathbb{N}^k$ there are finitely many $\sigma_1 + \sigma_2 = \sigma$ in \mathbb{N}^k .

In most literature this formal sum notation is standard and often useful because FPS arithmetic operations are very similar to how one would do calculations with “real” sums. The difference between *formal* sums and “real” sums are quite delicate. Particularly, convergence is of secondary importance, e.g., we oftentimes write $\frac{1}{1-X} = \sum_{n \in \mathbb{N}} X^n$, regardless of the fact whether $|X| < 1$. This is possible due to the underlying algebraic structure, i.e., Equation (1) can be interpreted as follows in the ring of FPS: The “sequences” $2 - 1X + 0X^2 + \dots$ and $1/2 + 1/4X + 1/8X^2 + \dots$ are (multiplicative) *inverse* elements to each other in $\mathbb{R}[[X]]$, i.e., their product is 1. More generally, we say that an FPS F is *rational* if $F = GH^{-1} = G/H$ where G and H are polynomials, i.e., they have at most finitely many non-zero coefficients. We call such a representation a *rational closed form*.

We emphasize that the indeterminates X are merely *labels* for the k dimensions of F and do not have any other particular meaning. However –inspired by Klinkenberg et al. [2020]– we naturally identify the indeterminates with the program variables (e.g. indeterminate X refers to variable x).

2.3 Probability Generating Functions

Towards our goal of analyzing posterior distributions described by probabilistic programs, we are especially interested in GF that describe discrete probability distributions.

Definition 2 (PGF [Chen et al. 2022]). A k -dimensional FPS G is a probability generating function (PGF) if (i) for all $\sigma \in \mathbb{N}^k$ we have $[\sigma]_G \geq 0$, and (ii) $\sum_{\sigma \in \mathbb{N}^k} [\sigma]_G \leq 1$.

Note that Definition 2 also includes *sub-PGF* where the sum in (ii) is strictly less than 1.

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 an algebraic structure: the ring of *formal power series* (FPS) [Chen et al. 2022] – 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, both for probabilities and for assignments 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 a finite number of \mathbb{N} -valued program variables x_1, x_2, \dots, x_k . The set of program states is therefore given by \mathbb{N}^k , where for each $\sigma = (\sigma_1, \dots, \sigma_k) \in \mathbb{N}^k$, σ_i indicates the value of x_i .

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

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

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

where E is an arithmetic expression, B is a Boolean expression, and $p \in [0, 1]$.

The (operational) meaning of most cpGCL program constructs are standard. The *probabilistic choice* $\{P\} [p] \{Q\}$ executes P with probability $p \in [0, 1]$ and Q with probability $1 - p$. The *conditioning statement* $\text{observe}(B)$ “blocks” all program runs that violate the guard B and normalizes the probabilities of the remaining runs. Recall Prog. 1 on page 2, which models the telephone operator scenario. In that example, the operator observes 5 calls in the last hour (indicated by $\text{observe}(c = 5)$). To reflect this, all program states where $c \neq 5$ are assigned probability zero, which may result in losing probability mass. We redistributed the lost probability mass over the valid program states by *renormalizing* the distribution appropriately. To identify program runs violating the observations, we extend the domain of formal power series – and thereby the domain of PGFs – with a dedicated indeterminate aggregating the probability of observation violations:

Definition 4 (eFPS and ePGF). Let $\mathbf{X} = (X_1, \dots, X_k)$ and X_ℓ be indeterminates. For any program state $\sigma = (\sigma_1, \dots, \sigma_k) \in \mathbb{N}^k$, \mathbf{X}^σ denotes the monomial $X_1^{\sigma_1} \cdots X_k^{\sigma_k}$. An extended formal power series (eFPS) is an object of the form

$$F = [\ell]_F X_\ell + \sum_{\sigma \in \mathbb{N}^k} [\sigma]_F \mathbf{X}^\sigma \quad \text{with} \quad [\cdot]_F: \mathbb{N}^k \cup \{\ell\} \rightarrow \mathbb{R}_{\geq 0}^\infty.$$

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

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

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

Remark. eFPS is not closed under multiplication, e.g., $X_\ell \cdot X_\ell = X_\ell^2 \notin \text{eFPS}$. This is in fact intended, as such monomial combinations do not have a meaningful interpretation in terms of probability distributions. \triangleleft

We endow eFPSs and ePGFs with the following ordering relations.

Definition 5 (Orders over eFPS). For all $F, G \in \text{eFPS}$, let

$$F \leq G \quad \text{iff} \quad \forall \sigma \in \mathbb{N}^k \cup \{\ell\}. [\sigma]_F \leq [\sigma]_G.$$

Additionally, for all $\phi, \psi \in (\text{eFPS} \rightarrow \text{eFPS})$, let

$$\phi \sqsubseteq \psi \quad \text{iff} \quad \forall F \in \text{eFPS}. \phi(F) \leq \psi(F).$$

To evaluate Boolean guards, we use the so-called *filtering* function for eFPSs. The filtering of $F \in \text{eFPS}$ by Boolean expression B is

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

i.e., $\langle F \rangle_B$ is the eFPS derived from F by setting $[\frac{1}{2}]_F$ 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 $F = \langle F \rangle_B + \langle F \rangle_{\neg B} + [\frac{1}{2}]_F X_\ell$. Further properties of the eFPS domain are found in Appendix B.

3.1 Unconditioned Semantics for cpGCL

Table 2. The *unconditioned* semantics for cpGCL programs; $\text{eval}_\sigma(E)$ is the evaluation of expression E in program state σ .

P	$\llbracket P \rrbracket(G)$
skip	G
$x_i := E$	$[\frac{1}{2}]_G X_\ell + \sum_\sigma [\sigma]_G X_1^{\sigma_1} \dots X_i^{\text{eval}_\sigma(E)} \dots X_k^{\sigma_k}$
observe (B)	$([\frac{1}{2}]_G + \langle G \rangle_{\neg B}) X_\ell + \langle G \rangle_B$
$\{P_1\} [p] \{P_2\}$	$p \cdot \llbracket P_1 \rrbracket(G) + (1 - p) \cdot \llbracket P_2 \rrbracket(G)$
if (B) $\{P_1\}$ else $\{P_2\}$	$[\frac{1}{2}]_G X_\ell + \llbracket P_1 \rrbracket(\langle G \rangle_B) + \llbracket P_2 \rrbracket(\langle G \rangle_{\neg B})$
$P_1 \mathbin{\text{;}} P_2$	$\llbracket P_2 \rrbracket(\llbracket P_1 \rrbracket(G))$
while (B) $\{P_1\}$	$\text{lf}_p \Phi_{B,P_1}(G)$, where $\Phi_{B,P_1}(f) = \lambda G. [\frac{1}{2}]_G X_\ell + \langle G \rangle_{\neg B} + f(\llbracket P_1 \rrbracket(\langle G \rangle_B))$

Let $\llbracket P \rrbracket: \text{eFPS} \rightarrow \text{eFPS}$ be an (unconditioned) distribution transformer for program P . We define the *unconditioned* semantics for a cpGCL program P by transforming an input eFPS G to an output eFPS $\llbracket P \rrbracket(G)$ while *explicitly* keeping track of the probability of violating the observations; see Table 2. With $G \in \text{ePGF}$ as an initial distribution, let us examine the semantics $\llbracket P \rrbracket(G)$:

The case “ $P = \text{skip}$ ” leaves the initial distribution G unchanged, i.e., it *skips* an instruction. Assignment “ $P = x_i := E$ ” semantics is defined using the auxiliary function $\text{eval}_\sigma(E)$, which *evaluates* a given expression E in a program state σ . For instance, given $E = 2 \cdot xy^3 + 23$ and state $(x, y) = (1, 10)$, $\text{eval}_\sigma(E) = 2 \cdot 10^3 + 23 = 2023$. Using the evaluation function, the assignment instruction updates every exponent of variable X_i state-wise by $\text{eval}_\sigma(E)$. In line with [Jacobs 2021; Nori et al. 2014; Olmedo et al. 2018], the semantics for “observe(B)” is defined as *rejection sampling*, i.e., if the current program run satisfies B , it behaves like a skip statement and the posterior distribution is unchanged. If the current run, however, violates the condition B , the run is rejected and the program restarts from the top in a reinitialized state. Hence, observing a certain guard B just filters the prior distribution and accumulates the probability mass that violates the guard. For example observing an even dice roll observe ($x \equiv_2 0$) out of a six sided die $\frac{1}{6}(X + X^2 + X^3 + X^4 + X^5 + X^6)$ yields $\frac{1}{6}(X^2 + X^4 + X^6) + \frac{1}{2}X_\ell$. Probabilistic branching “ $P = \{P_1\} [p] \{P_2\}$ ” is interpreted as the convex p -weighted combination of the two subprograms P_1 and P_2 . Conditional branching “ $P = \text{if } (B) \{P_1\} \text{ else } \{P_2\}$ ” is defined by conditionally combining the two subprograms P_1 and P_2 dependent on B . Sequential composition of programs “ $P_1 \mathbin{\text{;}} P_2$ ” composes programs in a *forward* manner, i.e., we first evaluate P_1 and take the intermediate result as new input for P_2 . Finally, we consider loops “ $P = \text{while } (B) \{P_1\}$ ” which are the most complicated expression in cpGCL. To define its semantics we rely on domain theory and the existences of fixed points, i.e., the semantics of a loop is given by the least fixed point of Φ_{B,P_1} . Here, Φ_{B,P_1} is called the characteristic function of the loop while (B) $\{P_1\}$ as it mimics the behavior of unfolding the loop. Concretely, Φ_{B,P_1} formalizes the equivalence of while (B) $\{P_1\}$ and if (B) $\{P_1 \mathbin{\text{;}} \text{while } (B) \{P_1\}\}$ else {skip}.

Note that the observe violation term 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 observe violation probabilities, both semantics coincide.

Recall Prog. 3 on page 3, where all program runs which eventually would terminate violate the observation. As 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 $1 \cdot X_{\frac{1}{2}}$.

3.2 Conditioned Semantics for cpGCL

The unconditioned semantics serves as an intermediate encoding to achieve our *conditioned* semantics, which further addresses *normalization* of distributions. We define the normalization operator of a eFPS by

Definition 6 (Normalization). *The normalization operator $norm$ is a partial function defined as²*

$$norm: \text{eFPS} \rightarrow \text{eFPS}, \quad F \mapsto \begin{cases} \frac{\langle F \rangle_{\text{true}}}{1 - [\frac{1}{2}]_F} & \text{if } [\frac{1}{2}]_F < 1, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Intuitively, normalizing an eFPS amounts to “distributing” the probability mass $[\frac{1}{2}]_F$ pertaining to observation violations over its remaining (valid) program runs. We lift the operator and denote the *conditioned* semantics of P by

$$norm(\llbracket P \rrbracket) \triangleq \lambda G. norm(\llbracket P \rrbracket(G)) = \lambda G. \frac{\langle \llbracket P \rrbracket(G) \rangle_{\text{true}}}{1 - [\frac{1}{2}]_{\llbracket P \rrbracket(G)}}.$$

Remark. In contrast to the unconditioned semantics, the conditioned semantics might not always be defined: Reconsider Prog. 3 for which the unconditioned semantics is $1 \cdot X_{\frac{1}{2}}$; normalizing the semantics is not possible as it would lead to $\frac{\langle 1 \cdot X_{\frac{1}{2}} \rangle_{\text{true}}}{1 - [\frac{1}{2}]_F} = \frac{0}{0}$, 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: $\{x := 1\} [\frac{1}{2}] \{ \text{observe (false)} \}$ has a conditioned semantics of $1 \cdot X^1$, whereas the conditioned semantics for $\{x := 1\} [\frac{1}{2}] \{ \text{diverge} \}$ ³ is $\frac{1}{2} \cdot X^1$. \triangleleft

Example 7 (Telephone Operator). Reconsider Prog. 1, the loop-free program generating a distribution with infinite support. It pertains to 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 $\frac{5}{7}$ probability of it being a weekday ($w = 0$) and a $\frac{2}{7}$ 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

```

1      ( = 1 · W0C0 + 0 · X1 )
{ w := 0 } [ 5/7 ] { w := 1 } ;
5/7 W0 + 2/7 W1
if ( w = 0 ) {
  5/7
  c := poisson (6)
  5/7 e-6(1-C)
} else {
  2/7 W
  c := poisson (2)
  2/7 e-2(1-C) W
} ;
5/7 e-6(1-C) + 2/7 e-2(1-C) W
observe ( c = 5 )
5/7 (4860+8e4W) C5 + (1 - 4860+8e4) X1

```

Prog. 5. Semantics for the tel. operator.

² $norm$ in fact maps an eFPS to an FPS, i.e., $[\frac{1}{2}]_F X_{\frac{1}{2}}$ is pruned away by normalization.

³diverge is syntactic sugar for while (true) { skip }.

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:

$$\llbracket P \rrbracket(G) = \frac{(4860+8e^4W)}{105e^6}C^5 + (1 - \frac{4860+8e^4}{105e^6})X_{\downarrow}.$$

Normalizing this unconditioned semantics yields

$$\text{norm}(\llbracket P \rrbracket(G)) = \frac{(1215e^{-4} + 2W)C^5}{2 + 1215e^{-4}}. \quad \triangleleft$$

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 programs⁴ 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 3 ingredients: (1) The state space \mathcal{S} ; (2) the initial state $\langle P, \sigma \rangle$ and (3) a transition matrix $\mathcal{P} : \mathcal{S} \times \mathcal{S}$. The states are pairs of the form $\langle P, \sigma \rangle$. Here, P denotes the program left to be executed, where \downarrow indicates the terminated program and σ the current state valuation. We use the dedicated state $\langle \downarrow \rangle$ for denoting that some observe violations have occurred during the run of a program. The detailed construction of the Markov chain for $\mathcal{R}_\sigma \llbracket P \rrbracket$ from a cpGCL program P with initial state σ is given in Appendix B. Regarding the equivalence between the two semantics, we are interested in the reachability probabilities of eventually reaching a state $\langle \downarrow, \sigma \rangle$ conditioned to never visiting the observe violation state $\langle \downarrow \rangle$.

Theorem 8 (Equivalence of Semantics). *For every cpGCL program P , let $\mathcal{R}_\sigma \llbracket P \rrbracket$ be the Markov chain of P starting in state $\sigma \in \mathbb{N}^k$. Then, for any $\sigma' \in \mathbb{N}^k$,*

$$\Pr^{\mathcal{R}_\sigma \llbracket P \rrbracket}(\diamond \langle \downarrow, \sigma' \rangle \mid \neg \diamond \langle \downarrow \rangle) = [\sigma']_{\text{norm}(\llbracket P \rrbracket(X^\sigma))},$$

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

This coincidence 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], since an observe-free program never induces the violation term $[\downarrow]_F X_{\downarrow}$ and hence, the *norm* operator has no effect.

4 EXACT BAYESIAN INFERENCE WITH LOOPS

Loops significantly complicate inferring posterior distributions in probabilistic programs. Computing the exact fixed point of the characteristic function $\Phi_{B,P}$ is often intractable, thus other approaches like invariant-based reasoning techniques are used. This section endeavors a program equivalence approach on restricted *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

In order to develop invariant-based reasoning techniques, we first have to clarify some notions about the termination behavior of probabilistic programs [Bournez and Garnier 2005; Saheb-Djahromi 1978].

⁴Programs that terminate with probability 1 on all inputs; see Definition 9.

Definition 9 (Almost-Sure Termination). A program P is almost-surely terminating (AST) for $F \in \text{eFPS}$ if

$$|\llbracket P \rrbracket(F)| + [\downarrow]_{\llbracket P \rrbracket(F)} = |F| + [\downarrow]_F.$$

Program P is universally AST (UAST) if it is AST for all $F \in \text{eFPS}$.

We approximate the least fixed point of the characteristic function for a loop in terms of applying domain theory, in particular Park's lemma [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 10 (Loop Invariants). Let $L = \text{while}(B)\{P\}$, and let I be a loop-free program.

- (1) $\Phi_{B,P}(\llbracket I \rrbracket) \subseteq \llbracket I \rrbracket$ implies $\llbracket L \rrbracket \subseteq \llbracket I \rrbracket$.
- (2) $\text{norm}(\llbracket L \rrbracket(F)) \leq \text{norm}(\llbracket I \rrbracket(F))$, whenever $\text{norm}(\llbracket I \rrbracket(F))$ is defined.
- (3) If L is UAST, (1) and (2) are equalities, i.e.,
 $\Phi_{B,P}(\llbracket I \rrbracket) = \llbracket I \rrbracket$ implies $\llbracket L \rrbracket = \llbracket I \rrbracket$ and $\text{norm}(\llbracket L \rrbracket(F)) = \text{norm}(\llbracket I \rrbracket(F))$.

PROOF.

Ad (1): Let $\Phi_{B,P}(\llbracket I \rrbracket) \subseteq \llbracket I \rrbracket$. From the second variant of the fixed point theorem in [Abramsky and Jung 1994; Lassez et al. 1982] it follows that

$$\begin{aligned} \llbracket \text{while}(B)\{P\} \rrbracket &= \text{lfp } \Phi_{B,P} \\ &= \inf \{ \psi \in (\text{eFPS} \rightarrow \text{eFPS}) \mid \Phi_{B,P}(\psi) \subseteq \psi \} \subseteq \llbracket I \rrbracket. \end{aligned} \quad \triangleleft$$

Ad (2): We first prove, that the normalization function is monotonic, whenever it is defined. Therefore, let $F, G \in \text{eFPS}$ such that $\text{norm}(F), \text{norm}(G)$ are defined.

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

Thus, whenever $\text{norm}(\llbracket I \rrbracket(F))$ is defined, $\text{norm}(\llbracket \text{while}(B)\{P\} \rrbracket(F)) \leq \text{norm}(\llbracket I \rrbracket(F))$ must hold, due to Theorem 10 (1). \triangleleft

Ad (3): Now suppose that $\Phi_{B,P}(\llbracket I \rrbracket) = \llbracket I \rrbracket$, i.e., $\llbracket I \rrbracket$ is a fixed point of $\Phi_{B,P}$. Then, $\llbracket I \rrbracket$ must be at least the least fixed point $\text{lfp } \Phi_{B,P} = \llbracket \text{while}(B)\{P\} \rrbracket$, due to Theorem 10 (1). If $\text{while}(B)\{P\}$ is UAST then $|\llbracket \text{while}(B)\{P\} \rrbracket(F)| + [\downarrow]_{\llbracket \text{while}(B)\{P\} \rrbracket(F)} = |F| + [\downarrow]_F = |\llbracket I \rrbracket(F)| + [\downarrow]_{\llbracket I \rrbracket(F)}$ for all $F \in \text{eFPS}$. The second equality arises from I being a loop-free program which are trivially UAST. Combining these results, yields:

$$\begin{aligned} \forall F \in \text{eFPS}. \llbracket \text{while}(B)\{P\} \rrbracket(F) &\leq \llbracket I \rrbracket(F) \\ \text{and } |\llbracket \text{while}(B)\{P\} \rrbracket(F)| + [\downarrow]_{\llbracket \text{while}(B)\{P\} \rrbracket(F)} &= |\llbracket I \rrbracket(F)| + [\downarrow]_{\llbracket I \rrbracket(F)} \\ \implies \forall F \in \text{eFPS}. \llbracket \text{while}(B)\{P\} \rrbracket(F) &= \llbracket I \rrbracket(F) \\ \iff \llbracket \text{while}(B)\{P\} \rrbracket &= \llbracket I \rrbracket \end{aligned}$$

Having $\text{norm}(\llbracket \text{while}(B)\{P\} \rrbracket(F)) = \text{norm}(\llbracket I \rrbracket(F))$ then follows for all $F \in \text{eFPS}$. \square

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

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

Effectively, reasoning about loops is reduced to *two challenges*: First, finding a invariant candidate program I and second, verifying that the candidate indeed is a valid invariant, i.e., deciding whether $\Phi_{B,P}(\llbracket I \rrbracket) \sqsubseteq \llbracket I \rrbracket$. In this section we focus on *verifying* given invariant candidates while deferring finding invariants to [Section 5](#).

4.2 Verification of Loop Invariants

Checking whether I is an invariant of $\text{while } (B) \{ P \}$ amounts to check:

$$\forall G \in \text{eFPS} \quad \forall \sigma \in \mathbb{N}^k \cup \{\ell\}: \quad [\sigma]_{\Phi_{B,P}(\llbracket I \rrbracket)(G)} \leq [\sigma]_{\llbracket I \rrbracket(G)},$$

that rephrases reasoning about loops to the domination of distributions described by loop-free programs. Unfortunately this problem is undecidable, hence we consider the simplified version

$$\forall G \in \text{eFPS} \quad \forall \sigma \in \mathbb{N}^k \cup \{\ell\}: \quad [\sigma]_{\Phi_{B,P}(\llbracket I \rrbracket)(G)} = [\sigma]_{\llbracket I \rrbracket(G)}. \quad (\dagger)$$

The latter considers the special case of two distributions being equal. In other words, we are interested in the equivalence of two loop-free programs. Regardless of program equivalence being undecidable, we introduce a syntactic restriction of loop-free cpGCL programs where program equivalence is decidable and which has non-trivial expressiveness. To that end, we characterize cpGCL by means of an alternative representation called cReDiP—conditional Rectangular Discrete Probabilistic programs. The full syntax and semantics of cReDiP is described in [Table 3](#).

Let us explain the important parts of cReDiP in detail: First, it suffices to consider $\text{observe}(\text{false})$ statements, as in combination with if-then-else branching, we can reconstruct the full expressiveness of rectangular guards for observe statements. However, it is important that we explicitly substitute only the variables X which occur in P by 1. Second, the $\text{iid}(D, y)$ operation sums y many independent samples of distribution D where D can be represented in terms of a rational closed-form PGF. This statement ensures the non-trivial expressiveness of the loop-free fragment as it also subsumes the possibility to sample from the specified distributions D . For example, the program $P := y := 10; x := 0; x += \text{iid}(\text{bernoulli}(1/2), y)$ describes a binomial distribution in X with parameters $n = 10$ and $p = 1/2$, i.e. $\llbracket P \rrbracket = Y^{10} \cdot (1/2 + 1/2X)^{10}$. Note that cReDiP is as expressive as cpGCL is (both are Turing complete), however their loop-free fragments differ.

The advantage of cReDiP is that closed-form rational distributions are preserved throughout the whole loop-free fragment, enabling us to reason about infinite support discrete distributions in a symbolic manner. The latter implies, that we can decide the equivalence of loop-free cReDiP programs using the following approach based on extended second order PGFs.

Definition 11 (Second-Order ePGF). Let $\mathbf{U} = (U_1, \dots, U_k)$ be a tuple of formal indeterminates, that are pairwise distinct from $\mathbf{X} = (X_1, \dots, X_k)$ and X_i of eFPS. A second-order ePGF is a generating function of the form

$$G = \sum_{\sigma \in \mathbb{N}^k} G_\sigma U^\sigma = \sum_{\sigma \in \mathbb{N}^k} (\langle G_\sigma \rangle_{\text{true}} + [\downarrow]_\sigma X_i) U^\sigma = \sum_{\sigma \in \mathbb{N}^k} \langle G_\sigma \rangle_{\text{true}} U^\sigma + \sum_{\sigma \in \mathbb{N}^k} [\downarrow]_\sigma X_i U^\sigma,$$

where $G_\sigma \in \text{ePGF}$. 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,

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

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 12.

Lemma 12 (Error Term Pass-Through). For every program P and every $F \in \text{eFPS}$, the error term $[\downarrow]_F X_i$ passes through the transformer unaffected, i.e.

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

PROOF. See Appendix B. □

Currently, the semantics of a cReDiP Program is defined over eFPS which is a subset of eSOP. However, eSOP can be embedded into eFPS by simply adding the additional indeterminates \mathbf{U} . This identification allows us to apply the denotational semantics $\llbracket \cdot \rrbracket$ also on eSOP elements in a meaningful way.

Theorem 13 (eSOP Semantics). Let P be a loop-free cReDiP Program. Let $G = \sum_{\sigma \in \mathbb{N}^k} G_\sigma \mathbf{U}^\sigma \in \text{eSOP}$. 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. See Appendix C. □

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 $\llbracket P \rrbracket(G)$ 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 14 (eSOP Characterization). Let P_1 and P_2 be loop-free cReDiP-programs with $\text{Vars}(P_i) \subseteq \{x_1, \dots, x_k\}$ for $i \in \{1, 2\}$. Further, consider a vector $\mathbf{U} = (U_1, \dots, U_k)$ of meta indeterminates, and let \hat{G} be the eSOP $(1 - X_1 U_1)^{-1} \cdots (1 - X_k U_k)^{-1} \in \mathbb{R}[[\mathbf{X}, \mathbf{U}]]$. Then,

$$\forall G \in \text{eFPS}. \llbracket P_1 \rrbracket(G) = \llbracket P_2 \rrbracket(G) \iff \llbracket P_1 \rrbracket(\hat{G}) = \llbracket P_2 \rrbracket(\hat{G}).$$

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

$$\begin{aligned}
& \llbracket P_1 \rrbracket(\hat{G}) = \llbracket P_2 \rrbracket(\hat{G}) \\
\iff & \llbracket P_1 \rrbracket(\hat{G}) - \llbracket P_2 \rrbracket(\hat{G}) = 0 \\
\iff & \llbracket P_1 \rrbracket\left(\sum_{\sigma \in \mathbb{N}^k} \mathbf{X}^\sigma \mathbf{U}^\sigma\right) - \llbracket P_2 \rrbracket\left(\sum_{\sigma \in \mathbb{N}^k} \mathbf{X}^\sigma \mathbf{U}^\sigma\right) = 0 \\
\iff & \sum_{\sigma \in \mathbb{N}^k} \llbracket P_1 \rrbracket(\mathbf{X}^\sigma) \mathbf{U}^\sigma - \sum_{\sigma \in \mathbb{N}^k} \llbracket P_2 \rrbracket(\mathbf{X}^\sigma) \mathbf{U}^\sigma = 0 & \text{(By Theorem 13)} \\
\iff & \sum_{\sigma \in \mathbb{N}^k} (\llbracket P_1 \rrbracket(\mathbf{X}^\sigma) - \llbracket P_2 \rrbracket(\mathbf{X}^\sigma)) \mathbf{U}^\sigma = 0 & \text{(rewriting)} \\
\iff & \forall \sigma \in \mathbb{N}^k: \llbracket P_1 \rrbracket(\mathbf{X}^\sigma) - \llbracket P_2 \rrbracket(\mathbf{X}^\sigma) = 0 & \text{(By definition of the 0-FPS in } \mathbb{R}[[\mathbf{X}, \mathbf{X}_t, \mathbf{U}]]\text{)} \\
\iff & \forall \sigma \in \mathbb{N}^k: \llbracket P_1 \rrbracket(\mathbf{X}^\sigma) = \llbracket P_2 \rrbracket(\mathbf{X}^\sigma) \\
\iff & \llbracket P_1 \rrbracket = \llbracket P_2 \rrbracket & \text{(by Kozen [1981] and Lemma 12)}
\end{aligned}$$

□

As we can compute $\llbracket P \rrbracket(G)$ for loop-free $P \in \text{cReDiP}$ the following consequence is immediate:

Corollary 15 (Decidability of Equivalence). *Let P, Q be two loop-free cReDiP programs. Then:*

$$\forall G \in \text{eFPS}. \llbracket P \rrbracket(G) = \llbracket Q \rrbracket(G) \text{ is decidable.}$$

PROOF. By utilizing Lemma 14, we can rephrase the problem of determining program equivalence through the eSOP characterization $\llbracket P_1 \rrbracket(\hat{G}) = \llbracket P_2 \rrbracket(\hat{G})$. 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 14). As \hat{G} is in rational closed form, both $\llbracket P_1 \rrbracket(\hat{G})$ and $\llbracket P_2 \rrbracket(\hat{G})$ must also possess a rational closed form (since cReDiP semantics preserve closed forms; see Table 3 and [Chen et al. 2022]). Additionally, the effective computation of $\llbracket P_1 \rrbracket(\hat{G}) = \frac{F_1}{H_1}$ and $\llbracket P_2 \rrbracket(\hat{G}) = \frac{F_2}{H_2}$ is possible because both P_1 and P_2 are assumed to be loop-free programs.

In $\mathbb{R}[[\mathbf{X}, \mathbf{X}_t, \mathbf{U}]]$, 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:

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

since the latter equation concerns the equivalence of two polynomials in $\mathbb{R}[[\mathbf{X}, \mathbf{X}_t, \mathbf{U}]]$. 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., $\llbracket P_1 \rrbracket = \llbracket P_2 \rrbracket$), 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 of $F_1H_2 - F_2H_1$ and computing the first non-zero coefficient in $\mathbb{R}[[\mathbf{X}, \mathbf{X}_t]]$. Afterwards, extracting the exponent of the monomial describes an initial program state σ , where $\llbracket I \rrbracket(\sigma)$ and $\Phi_{B,P}(\llbracket I \rrbracket)(\sigma)$ do not coincide. Note that this only is a counterexample *to induction* as this does not mean, that I and W differ on this input. □

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

Theorem 16. *Let $L = \text{while } (B) \{P\} \in \text{cReDiP}$ be UAST with loop-free body P and I be a loop-free cReDiP-program. It is decidable whether $\llbracket L \rrbracket = \llbracket I \rrbracket$.*

PROOF. The correctness is an immediate consequence of [Theorem 10](#) and [Corollary 15](#). \square

We demonstrate the invariant reasoning technique in [Example 17](#).

<pre> while (y = 1) { { y := 0 } [1/2] { y := 1 } ; x := x + 1 ; observe (x < 3) } </pre>	<pre> if (y = 1) { x += iid (geom(1/2), y) ; y := 0 ; observe (x < 3) } </pre>
--	---

Prog. 6. The truncated geometric distribution generator.

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

Example 17 (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 $1 \cdot Y^1 X^0$. We first evaluate the least fixed point of $\Phi_{B,P}$. Using [Theorem 10](#) (3), 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 $\llbracket P \rrbracket(Y) = \frac{4}{7} + \frac{2}{7}X + \frac{1}{7}X^2$. 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](#). \triangleleft

To summarize, reasoning about program equivalence using eSOPs enables exact Bayesian inference for cReDiP programs containing loops. Our current 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 $\llbracket L \rrbracket = \llbracket I \rrbracket \implies \text{norm}(\llbracket L \rrbracket) = \text{norm}(\llbracket I \rrbracket)$.

4.3 Equivalence of Conditioned Semantics

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

$$P \sim Q \quad \text{iff} \quad \forall G \in \text{eFPS}. \text{norm}(\llbracket P \rrbracket(G)) = \text{norm}(\llbracket Q \rrbracket(G)) \quad (\ddagger)$$

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

Definition 18 (Conditioning on eSOP). Let $G \in \text{eSOP}$. The function

$$\text{cond}: \text{eSOP} \rightarrow \text{SOP}, \quad G \mapsto \sum_{\sigma \in \mathbb{N}^k} \text{norm}(G_\sigma) \mathbf{U}^\sigma.$$

is called the conditioning function.

For simplicity, we assume that $\forall \sigma \in \mathbb{N}^k. G_\sigma \neq X_\sharp$ as otherwise *norm* is not defined. Note that *cond* often *cannot* be evaluated in a closed-form eSOP because 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 19. Let $F_1, F_2 \in \text{ePGF}$, with $p := [\sharp]_{F_1} = [\sharp]_{F_2}$. Then:

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

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 observation:

Corollary 20 (Partitioning). *Let S be a finite partitioning of $\mathbb{N}^k = S_1 \uplus \dots \uplus S_m$ with $[\downarrow]_{G_\sigma} = [\downarrow]_{G_{\sigma'}}$, for all $\sigma, \sigma' \in S_i$, $1 \leq i \leq m$. Then:*

$$G = \sum_{i=1}^m \sum_{\sigma \in S_i} ([\downarrow]_{S_i} X_{\downarrow} + \langle G_\sigma \rangle_{\text{true}}) U^\sigma, \quad \text{where}$$

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

$$\text{cond}(G) = \sum_{i=1}^m \frac{\sum_{\sigma \in S_i} \langle G_\sigma \rangle_{\text{true}} U^\sigma}{1 - [\downarrow]_{S_i}}.$$

Unfortunately, there exist already loop-free programs, for which a finite partitioning is impossible. An example for such a program is provided in Prog. 8. Given an initial distribution for program 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 success 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 $\sum_{i=0}^{\infty} \frac{(2^{-i} + (1-2^{-i})X_{\downarrow})V^i}{(1-U)}$. 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 $\text{cond}(\llbracket P \rrbracket(\hat{G})) = \text{cond}(\llbracket Q \rrbracket(\hat{G}))$. This further implies $\forall \sigma \in \mathbb{N}^k$. $\text{norm}(\llbracket P \rrbracket(X^\sigma)) = \text{norm}(\llbracket Q \rrbracket(X^\sigma))$, i.e., equivalence on point-mass distributions. However, we do not necessarily have the precise equivalence as per Equation (§), as the *norm* operator used to define *cond* is a non-linear function⁵ and thus the point-mass distributions cannot be combined in a sensible way. Nevertheless, 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.

```

// (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);
// 2(1 - X↓) / ((1 - U)(2 - V)) + X↓ / ((1 - U)(1 - V))
// ∑i=0∞ (2-i + (1 - 2-i)X↓) Vi / (1 - U)

```

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

5 FINDING INVARIANTS USING PARAMETER SYNTHESIS

In contrast to the previous section which aims at *validating a given invariant*, in this section, we address the problem of *finding* such invariants. In general, the invariant synthesis problem is stated as follows: Given a while-loop L , find a mapping $I: \text{ePGF} \rightarrow \text{ePGF}$ such that $\Phi_{B,P}(I) \subseteq I$. 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 second order ePGFs to achieve decidability results for a subclass of invariant candidates. Ideally, we aim for expressions of invariants that are either in a parametric form, e.g., for a bivariate uniform distribution transformer a formula

⁵For the unconditioned semantics, general equivalence $\llbracket P \rrbracket = \llbracket Q \rrbracket$ follows from the linearity of the transformer.

like $I(x^a y^b) = \frac{1}{ab} \cdot (\sum_{i=1}^a x^i) \cdot (\sum_{i=1}^b y^i) = \frac{1}{ab} \frac{x^a - x}{1-x} \frac{y^b - y}{1-y}$, or can be described by a loop-free cReDiP Program Q such that $\llbracket Q \rrbracket = I$. Note that the second approach is a sufficient condition for I being expressible in a parametric form. The reverse direction, that every parametric expression can be expressed by a loop-free cReDiP program is not always true which is further investigated in [Section 7](#). For the rest of this section we focus on the latter approach of finding loop-free Programs describing invariants.

5.1 Synthesizing Parameters in Program Templates

Sometimes, the general shape of a posterior distribution is derivable from the context, but the precise parameters are tricky to tune. This situation can be modeled by specifying an invariant program Q_p , where the distribution parameters are symbolic. We illustrate the idea by [Example 21](#).

<pre>while (n > 0) { { n := n - 1 } [q/3] { c := c + 1 } }</pre>	<pre>c += iid (geom(p), n); n := 0</pre>
---	--

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

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

Example 21 (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 changed from $\frac{1}{2}$ to $\frac{q}{3}$, where q is a symbolic parameter. Combining these changes, it seems natural that this program might encode the n -fold geometric distribution⁶ with individual success probability $\frac{q}{3}$. Applying that prior knowledge, we formulate an invariant template given in Prog. 10, where c is a sum of n geometric distributions with a not further specified parameter p . Using [Theorem 10](#), we can derive the equivalence of Prog. 9 and Prog. 10 and obtain an equation in p and q :

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

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

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

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 substituting $p = \frac{q}{3}$ in our invariant template yields equivalence of these two programs. \triangleleft

This approach works in general as the following theorem describes:

Theorem 22 (Decidability of Parameter Synthesis). *Let W be a cReDiP while loop and Q_p be a loop-free cReDiP program. It is decidable whether there exist parameter values \mathbf{p} such that the instantiated template Q_p is an inductive invariant, i.e.,*

$$\exists \mathbf{p} \in \mathbb{R}^I. \forall \mathbf{X} \in \text{Vars}(W). \quad \llbracket W \rrbracket = \llbracket Q_p \rrbracket.$$

PROOF. The proof is a variant of [Corollary 15](#). Full details are provided in [Appendix D](#). \square

⁶Sometimes also called negative binomial distribution

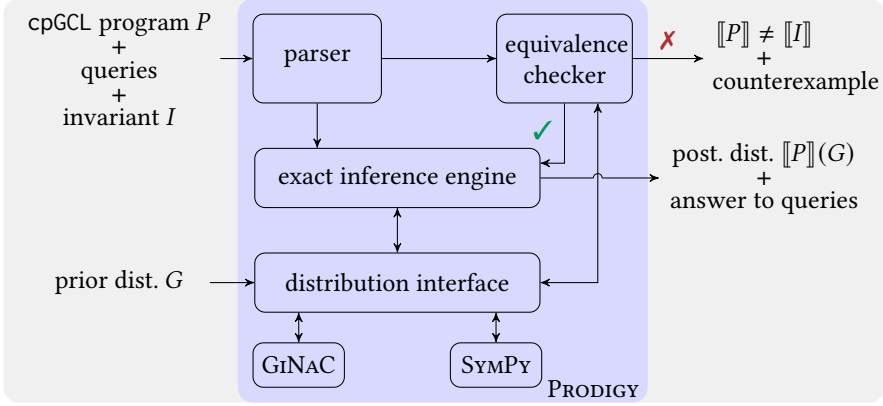


Fig. 3. A sketch of the PRODIGY workflow.

Note that in this formulation, parameters may depend on other parameters, but are always *independent* of all other program variables and second-order indeterminates.

6 EMPIRICAL EVALUATION OF PRODIGY

We have implemented our approach in Python as an extension to PRODIGY⁷[Chen et al. 2022] – Probability Distributions via Generatingfunctionology. The current project consists of about 6,000 LOC. The two main features we extended, are the implementation of 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, moments, which can be encoded inside of P) 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 also capable of computing under-approximations of the posterior distribution by unfolding the loop up to a specified accuracy or number of loop unrollings.

6.2 Benchmarks

We collected a set of 31 benchmarks. This set is composed of examples provided by λ -PSI [Gehr et al. 2020] and PRODIGY itself. All experiments were performed on MacOS Ventura 13.2 with a 2,4 GHz Quad-Core Intel Core i5 and 16GB RAM on a collection of (in)finite-state probabilistic programs. For

⁷<https://github.com/LKlinke/Prodigy>

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

Program	∞	p	PRODIGY		λ PSI
			SYMPY	GiNAC	
burgler_alarm			1.988	0.012	0.062
caesar		●	8.377	0.025	1.221
dnd_handicap			7.760	0.032	0.088
evidence1			0.348	0.002	0.012
evidence2			0.413	0.003	0.016
function			0.338	0.002	0.001
fuzzy_or			67.048	0.227	9.594
grass			6.706	0.021	0.471
infer_geom_mix		●	13.723	0.031	0.220
lin_regression_unbiased			6.700	0.014	0.059
lucky_throw			Err. ⁹	1.560	69.563
max			0.618	0.005	0.022
monty_hall			2.927	0.033	0.065
monty_hall_nested			15.694	0.140	0.473
murder_mystery		●	0.615	0.004	0.020
pi			90.931	0.094	TO
piranha			0.379	0.003	0.012
telephone_operator		●	1.249	0.006	0.064*
telephone_operator_param		●	5.880	0.017	0.108*
twocoins			0.493	0.004	0.013

each benchmark, we run PRODIGY with both CAS backends, i.e., SYMPY and GiNAC. The loop-free benchmarks are additionally compared against the timing results for λ -PSI, which is the closest tool out of all mentioned in Section 8. 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 only measure the time used for performing inference (compute the posterior distribution), i.e., the time spent for parsing the program, reading files, etc. is excluded⁸. Throughout the experiments we aim to answer questions in terms of (1) *Effectiveness*: Can PRODIGY effectively address the exact inference problem for the selected set of benchmarks including equivalence checking and invariant synthesis for programs with loops? (2) *Efficiency*: How does PRODIGY compare to λ -PSI? How do the integrated CAS back-ends compare to each other? Additionally, a comparison on the loop-free benchmark fragment without parameters against the implementation by Zaiser et al. [2023] would be desirable as they also base their approach on generating functions. Unfortunately their tool is not yet publicly available.

6.2.1 General observations. Our approach is able to compute posterior distributions on a variety of programs in less than 0.1 seconds. Considering the chosen benchmarks, *exact* Bayesian inference based on generating functions outperforms (up to several orders of magnitude) λ -PSI on *discrete* probabilistic programs. Even for models with possibly unbounded looping behavior and parameters, PRODIGY is able (provided with a suitable invariant) to compute the semantics thus inferring correct posterior distributions.

⁸We have extended the file `symbolic.d` of λ -PSI to precisely measure these timings.

⁹Reached maximum recursion limit

Table 5. Benchmarks for the inference of loopy probabilistic programs (some with parameter synthesis); timings are given in seconds.

Program	SYMPY	GiNAC
dep_bern	18.168	0.457
dueling_cowboys	9.654	0.090
geometric	5.588	0.039
geometric_parameter	17.687 $p = \frac{1}{3}$	0.293 $p = \frac{1}{3}$
n_geometric	4.667	0.052
n_geometric_parameter	10.074 $p = \frac{q}{3}$	0.310 $p = \frac{q}{3}$
random_walk	7.009	0.052
random_walk_parameter	10.165 $p = \frac{1}{2}$	0.298 $p = \frac{1}{2}$
trivial_iid	11.502	0.087
bit_flip_conditioning	TO	0.475
bit_flip_conditioning_parameter	TO	0.887 $p = \frac{13}{28}, q = \frac{3}{7}, r = \frac{2}{7}$

6.2.2 Results for loop-free programs. Table 4 depicts the quantitative results for loop-free programs. The column Program lists the benchmarks. The next column (∞) marks the occurrence of samplings from infinite-support distributions in the benchmark. Column p indicates the presence of symbolic parameters. Finally, columns SYMPY, GiNAC, and λ -PSI depict run-times in seconds for the individual backend of PRODIGY and λ -PSI, 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. Timings marked with * refer to results by λ -PSI which contain integral expressions. Strictly speaking, these results cannot be counted as “solved”, however we still considered them. Clearly, PRODIGY with its GiNAC backend is the clear winner in terms of run-time performance, however λ -PSI timings are oftentimes in a reasonable margin. Compared to the SYMPY backend, GiNAC is faster by roughly two orders of magnitude.

6.2.3 Results for loopy programs. 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 like to support the user in finding such invariants by allowing symbolic parameters for distributions, e.g., one can write $\text{geom}(p)$ where p is a symbolic parameter.

The structure of Table 5 is similar to the previous one, but for this set of benchmark programs, we do not compare to λ -PSI, as λ -PSI does neither support unbounded loops nor recursion. In some of the benchmarks for loops, we also provide some anticipated parameter constraints derived from the described approach. These examples are indicated by an inferred parameter value, which was automatically computed by PRODIGY. Whenever this is the case, we point out that for the GiNAC timings, discharging the resulting equation system is achieved using SYMPY solvers, which is due to missing functionality of GiNAC being able to solve these equation systems. Again, we see similar results to the previous benchmarks, i.e., GiNAC being faster by a factor of 100.

7 LIMITATIONS OF EXACT INFERENCE USING EFPS

We discuss some known limitations of the presented inference approach.

7.1 Simple Programs Encoding Complex Distributions

Oftentimes, simple programs can already encode complex distributions. For a concrete example, consider Prog. 11 which 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. Note that for any fixed program state $(0, 0) \neq (m, n) \in \mathbb{N}^2$,

the number of loop iterations is bounded by $n + m - 1$. Approaches like the ert-calculus [Kaminski et al. 2018] can easily verify upper bounds on the expected run-time for this program, but instead, we are interested in a more involved question about the exact posterior distribution of this program. 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 $\llbracket P \rrbracket(m^a n^b) = \sum_{i=1}^a \frac{m^i}{2^{a+b-i}} \cdot \binom{a+b-i-1}{b-1} + \sum_{i=1}^b \frac{n^i}{2^{a+b-i}} \cdot \binom{a+b-i-1}{a-1}$.

The distribution is in a summation form but usually we strive for closed-form descriptions. Using the simplification function in Mathematica [Inc. 2023], we are able to derive such a closed-form, namely

$$I(a, b) := 2^{1-a-b} m \binom{-2+a+b}{-1+b} {}_2F_1(1, 1-a, 2-a-b, 2m) + 2^{1-a-b} n \binom{-2+a+b}{-1+a} {}_2F_1(1, 1-b, 2-a-b, 2n).$$

Here ${}_2F_1$ denotes the hypergeometric function¹⁰. It shows, that the distribution is in some sense linked to the hypergeometric distribution, indicated by the occurrence of ${}_2F_1$ terms. Even though that function is quite complex, taking derivatives in m or n respectively is easy, 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. Unfortunately, we are not aware of any loop-free cReDiP invariant program, that generates this closed-form distribution. However, the GF semantics still enables us to proof 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.

7.2 Closed-Form Operations

Whenever we deal with infinite-support distributions, we rely heavily on operations computable on closed-forms that mimic the semantics of the statement. For example, given a joint distribution $G(X, Y)$ computing the semantics $\llbracket x := x * y \rrbracket(G(X, Y))$ is challenging. The current strategies try to figure out whether one of the marginal distributions is finite and compute the operation state-by-state. This however is very limited, as if both variables have an infinite support, we can only under-approximate the posterior distribution by truncating one of them. Developing sufficient conditions and closed-form operations would have an enormous impact on the range of GF applicability for exact Bayesian inference of probabilistic programs.

Another challenge is guard evaluation, i.e., filtering out the corresponding terms of a formal power series. It is somewhat surprising that our performance is proportional to the size of constants in the programs. Assume for instance a guard $x > n$, where n is just a constant. For larger n , the closed-form operation of computing the n -th formal derivative takes an increasing amount of time. Fortunately, GiNAC can cope well with “large” constants ($\sim 10^3$).

In case we are interested in the relation between two variables (like $x = y$), we have trouble evaluating this in closed-form. As before, by now we try to marginalize hoping that one of the variables is only finitely distributed and do the filtering state-wise. Unfortunately, when both

¹⁰More about this closed-form and algorithms to compute closed-forms alike can be found in [Petkovsek et al. 1996].

variables have marginal distributions with infinite support, we can again only under-approximate the *exact* posterior distribution. An interesting example why one cannot even strive for such a potential closed-form operation is Prog. 12. For this program, its variable r evaluates to 1 with probability $\frac{1}{\pi}$ after termination [Flajolet et al. 2011]. It thus might be interesting what syntactic restrictions *exactly capture algebraic* closed-forms.

```

 $x := \text{geom}(1/4) ;$ 
 $y := \text{geom}(1/4) ;$ 
 $t := x + y ;$ 
 $\{ t := t + 1 \} [5/9] \{ \text{skip} \} ;$ 
 $r := 1 ;$ 
loop(3){
   $s := \text{iid}(\text{bernoulli}(1/2), 2t) ;$ 
  if ( $s \neq t$ ) {  $r := 0$  }
}

```

Prog. 12. Non-algebraic Probabilities.

```

 $x := \text{geom}(1/2) ;$ 
loop( $n$ ){
  if ( $x \equiv 0 \pmod{2}$ ) {
     $\{ x := 3 * x + 1 \} [1/10]$ 
     $\{ x := 1/2 * x \}$ 
  } else {
     $x := 3 * x + 1$ 
  }
}

```

Prog. 13. Probabilistic Collatz's.

7.3 PRODIGY Scalability

Prog. 13 models a variant of the famous Collatz algorithm [Andrei and Masalagiu 1998]. The Collatz conjecture states that for all positive integers m there exists $n \in \mathbb{N}$ such that for the Collatz function $C(m) := n/2$ for $n \equiv (0 \pmod{2})$ 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.

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 by means of arithmetic progressions. Thus substituting complex valued roots of unity, makes the expression harder to process for the CAS. For this particular instance we could evaluate the guard by means of $\frac{G(X)+G(-X)}{2}$ to avoid the usage of complex numbers.

8 RELATED WORK

Probabilistic inference has undergone a recent surge of interest due to its vital role in modern probabilistic programming. We review a (far-from-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) bounds on expectations, called *quantitative loop invariants* [McIver and Morgan 2005]. There are a wide spectrum 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], in particular, leveraging *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. 2022] 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 program 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. Moreover, their exactness criterion is modulo floating-point arithmetic. This can sometimes lead to numerical errors as observed in their experiments. Opposite to that, our implementation uses exact arithmetic. Unfortunately, a comparison on the loop-free fragment is not possible as the source code of their tool is not yet publicly available.

As an alternative to PGFs, many probabilistic systems employ *probability density function* (PDF) representations of distributions, e.g., (λ)PSI [Gehr et al. 2016, 2020], AQUA [Huang et al. 2021] and HAKARU [Narayanan et al. 2016], as well as the density compiler in [Bhat et al. 2012, 2017]. These systems are dedicated to inference for programs encoding joint (discrete-)continuous distributions with conditioning. Reasoning about the underlying PDF representations, however, amounts to resolving complex integral expressions in order to answer inference queries, thus confining these techniques either to (semi-)numerical methods [Bhat et al. 2012, 2017; Huang et al. 2021; Narayanan et al. 2016] or exact methods yet limited to bounded looping behaviors [Gehr et al. 2016, 2020]. 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 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. 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 outperforms state-of-the-art inference tools over benchmarks of loop-free programs.

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 Schröder. 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 Schröder. 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 Česka, 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. 2022. Does a Program Yield the Right Distribution? Verifying Probabilistic Programs via Generating Functions. In *CAV (1) (LNCS, Vol. 13371)*. Springer, 79–101.
- 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.
- 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. λ 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, 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.
- T. Minka, J.M. Winn, J.P. Guiver, Y. Zaykov, D. Fabian, and J. Bronskill. 2018. *Infer.NET 0.3*. 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).
- M. Petkovsek, H.S. Wilf, and D. Zeilberger. 1996. *A = B*. CRC Press. <https://books.google.de/books?id=5UBZDwAAQBAJ>
- 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.
- N. 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.
- 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).
- J. Vellinga. 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 Luke Ong. 2023. Exact Bayesian Inference on Discrete Models via Probability Generating Functions: A Probabilistic Programming Approach. *CoRR* abs/2305.17058 (2023).

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 ∞ , 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 $(D \rightarrow D')$ as the set of functions $\{f: D \rightarrow D'\}$. We write vectors in bold-face notations like \mathbf{X} for (X_1, \dots, X_k) and $\mathbf{1} = (1, \dots, 1)$ 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 $\partial_{\mathbf{x}}^{\mathbf{I}} f := \frac{\partial^{\mathbf{I}} f}{\partial \mathbf{x}^{\mathbf{I}}}$.

Definition 23 (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 24 (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 25 (Monotonic Function). Let (D, \sqsubseteq) and (D', \sqsubseteq') be complete lattices. A function $f: D \rightarrow D'$ is monotonic if and only if:

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

Definition 26 (Continuous Functions). Let (D, \sqsubseteq) and (D', \sqsubseteq') be complete lattices. A function $f: D \rightarrow D'$ is Scott-continuous if and only if for every ω -chain, i.e., every set $S = \{s_n \mid n \in \mathbb{N}\} \subseteq D$ such that $s_0 \sqsubseteq s_1 \sqsubseteq s_2 \sqsubseteq \dots$, it holds that:

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

Lemma 27 (Continuous Functions are Monotone). Let (D, \sqsubseteq) and (D', \sqsubseteq') be complete lattices, and $f: D \rightarrow D'$ be a continuous function. Then f is monotonic.

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

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

□

Lemma 28 (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 $f, g \in (D \rightarrow D')$, $f \sqsubseteq' g$ if and only if $\forall d \in D. f(d) \leq' g(d)$. Then, $(D \rightarrow D', \sqsubseteq')$ is a partial order.

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

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

Lemma 29 (Point-Wise Lifting of Complete Lattices). *Let (D', \leq') be a complete lattice and \sqsubseteq' be a point-wise lifting of \leq' , i.e. for an arbitrary domain D and any $f, g \in (D \rightarrow D')$, let $f \sqsubseteq' g$ if and only if $\forall f \in D. f(d) \leq' g(d)$. Then $(D \rightarrow D', \sqsubseteq')$ is a complete lattice.*

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

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

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

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

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

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

Theorem 30 (Fixed Point Theorems [Abramsky and Jung 1994; Lassez et al. 1982]). *Let $f: D \rightarrow D$ be a continuous function on a complete lattice (D, \sqsubseteq) . Then f possesses a least fixed point denoted $\text{lfp } f$, which is given by:*

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

B SEMANTICS USING EFPS

Corollary 31 (Partial Orders over eFPS). *(eFPS, \leq) as well as the point-wise lifting on functions $(\text{eFPS} \rightarrow \text{eFPS}, \sqsubseteq)$ are partial orders.*

PROOF. Consider the coefficient function $[\cdot]_F \in (\mathbb{N}^k \cup \{\perp\} \rightarrow \mathbb{R}_{\geq 0}^\infty)$ which uniquely determines the eFPS F . We think of the order \leq as acting on the domain $(\mathbb{N}^k \cup \{\perp\} \rightarrow \mathbb{R}_{\geq 0}^\infty)$. Thus, \leq can be interpreted as the point-wise lifting of the (total) order \leq on $\mathbb{R}_{\geq 0}^\infty$, i.e., (eFPS, \leq) is a partial order by applying [Lemma 28](#). Since \sqsubseteq is a point-wise lifting of \leq , we can argue analogously for $(\text{eFPS} \rightarrow \text{eFPS}, \sqsubseteq)$. \square

Corollary 32 (Complete Lattices over eFPS). *Both partial orders (eFPS, \leq) and $(\text{eFPS} \rightarrow \text{eFPS}, \sqsubseteq)$ are complete lattices.*

PROOF. Analogously to the proof of [Corollary 31](#) we note that \leq is a point-wise lifting of \leq on $\mathbb{R}_{\geq 0}^\infty$, and \sqsubseteq is a point-wise lifting on \leq . Therefore applying [Lemma 29](#) twice yields the claimed result. \square

Lemma 33 (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 $(\text{eFPS} \rightarrow \text{eFPS}, \sqsubseteq)$.*

PROOF.

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

\square

Lemma 34 (Continuity of Auxiliary Functions). *For all $\sigma \in \mathbb{N}^k \cup \{\perp\}$ 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 $S = \{F_i \mid i \in \mathbb{N}\} \subseteq \text{eFPS}$ be an ω -chain with $F_0 \leq F_1 \leq F_2 \leq \dots$. Then:

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

\square

Theorem 35 (Continuity of $\llbracket \cdot \rrbracket$). *For every cpGCL program P , $\llbracket P \rrbracket$ is continuous on the domain $(\text{eFPS} \rightarrow \text{eFPS})$.*

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

Case $P = \text{skip}$:

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

Case $P = x_i := E$:

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

Case $P = \text{observe } (B)$:

$$\begin{aligned} \llbracket P \rrbracket(\sup S) &= (\llbracket \dot{x} \rrbracket_{\sup S} + |\langle \sup S \rangle_{-B}|) \cdot X_{\dot{x}} + \langle \sup S \rangle_B \\ &= (\llbracket \dot{x} \rrbracket_{\sup S} + \sup \{|\langle F \rangle_{-B}| \mid F \in S\}) \cdot X_{\dot{x}} \\ &\quad + \sup \{\langle F \rangle_B \mid F \in S\} \quad (\text{Cont. of } |\cdot|, \langle \cdot \rangle_B) \\ &= \sup \{(\llbracket \dot{x} \rrbracket_F + |\langle F \rangle_{-B}|) \cdot X_{\dot{x}} + \langle F \rangle_B\} \\ &= \sup \{\llbracket P \rrbracket(F) \mid F \in S\} \end{aligned}$$

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

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

Case $P = \text{if } (B) \{P_1\} \text{ else } \{P_2\}$:

$$\begin{aligned} \llbracket P \rrbracket(\sup S) &= \llbracket \dot{x} \rrbracket_{\sup S} \cdot X_{\dot{x}} + \llbracket P_1 \rrbracket(\langle \sup S \rangle_P) + \llbracket P_2 \rrbracket(\langle \sup S \rangle_{-B}) \\ &= \llbracket \dot{x} \rrbracket_{\sup S} \cdot X_{\dot{x}} + \sup \{\llbracket P_1 \rrbracket(\langle F \rangle_P) \mid F \in S\} \\ &\quad + \sup \{\llbracket P_2 \rrbracket(\langle F \rangle_{-B}) \mid F \in S\} \quad (\text{I.H. on } P_1 \text{ and } P_2) \\ &= \sup \{\llbracket \dot{x} \rrbracket_F \cdot X_{\dot{x}} + \llbracket P_1 \rrbracket(\langle F \rangle_P) + \llbracket P_2 \rrbracket(\langle F \rangle_{-B}) \mid F \in S\} \\ &= \sup \{\llbracket P \rrbracket(F) \mid F \in S\} \end{aligned}$$

Case $P = P_1 \circ P_2$:

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

Case $P = \text{while } (B) \{ P_1 \}$: In this case, we use that for all $n \in \mathbb{N}$, $\Phi_{B,P_1}^n(\perp)$ is continuous, which we prove by induction:

Base case: $n = 0$.

$$\Phi_{B,P_1}^0(\perp)(\sup S) = 0 = \sup \{ \Phi_{B,P_1}^0(\perp)(F) \mid F \in S \}$$

Induction step:

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

With this, it follows:

$$\begin{aligned} \llbracket P \rrbracket(\sup S) &= \left(\sup_{n \in \mathbb{N}} \Phi_{B,P_1}^n(\perp) \right) (\sup S) \\ &= \sup \{ \Phi_{B,P_1}^n(\perp)(\sup S) \mid n \in \mathbb{N} \} \\ &= \sup \{ \sup \{ \Phi_{B,P_1}^n(\perp)(F) \mid F \in S \} \mid n \in \mathbb{N} \} \quad (\text{Cont. of } \Phi_{B,P_1}^n(\perp)) \\ &= \sup \{ \sup \{ \Phi_{B,P_1}^n(\perp)(F) \mid n \in \mathbb{N} \} \mid F \in S \} \quad (\text{swap suprema}) \\ &= \sup \{ \sup \{ \Phi_{B,P_1}^n(\perp) \mid n \in \mathbb{N} \} (F) \mid F \in S \} \\ &= \sup \{ \llbracket P \rrbracket(F) \mid F \in S \} \end{aligned}$$

□

Lemma 36 (Linearity of Auxiliary Functions). *For all $\sigma \in \mathbb{N}^k \cup \{\downarrow\}$, $\alpha \in \mathbb{R}_{\geq 0}^\infty$, $F, G \in \text{eFPS}$ 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:

$$\begin{aligned} \alpha F + G &= \alpha \cdot \left([\downarrow]_F \cdot X_{\downarrow} + \sum_{\sigma \in \mathbb{N}^k} [\sigma]_F \cdot X^\sigma \right) \\ &\quad + \left([\downarrow]_G \cdot X_{\downarrow} + \sum_{\sigma \in \mathbb{N}^k} [\sigma]_G \cdot X^\sigma \right) \\ &= (\alpha \cdot [\downarrow]_F + [\downarrow]_G) \cdot X_{\downarrow} + \sum_{\sigma \in \mathbb{N}^k} (\alpha \cdot [\sigma]_F + [\sigma]_G) \cdot X^\sigma \quad (\#) \end{aligned}$$

By **Definition 4**:

$$\alpha F + G = [\downarrow]_{\alpha F + G} \cdot X_{\downarrow} + \sum_{\sigma \in \mathbb{N}^k} [\sigma]_{\alpha F + G} \cdot \mathbf{X}^\sigma \quad (\text{b})$$

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

(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:

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

□

Lemma 37 ($\Phi_{B,P}$ Preserves Linearity). *Let $\psi: \text{eFPS} \rightarrow \text{eFPS}$ be a linear function. If $\llbracket P \rrbracket$ is linear, then $\Phi_{B,P}(\psi)$ is linear as well.*

PROOF.

$$\begin{aligned} &\Phi_{B,P}(\psi)(\alpha F + G) \\ &= (\lambda F. [\downarrow]_F X_{\downarrow} + \langle F \rangle_{-B} + \psi(\llbracket P \rrbracket(\langle F \rangle_B))) (\alpha F + G) \\ &= [\downarrow]_{\alpha F + G} X_{\downarrow} + \langle \alpha F + G \rangle_{-B} + \psi(\llbracket P \rrbracket(\langle \alpha F + G \rangle_B)) \\ &= (\alpha [\downarrow]_F + [\downarrow]_G) \cdot X_{\downarrow} + \alpha \langle F \rangle_{-B} + \langle G \rangle_{-B} + \psi(\llbracket P \rrbracket(\alpha \langle F \rangle_B + \langle G \rangle_B)) \quad (\text{Lin. of } \langle \cdot \rangle_B \text{ (Lemma 36)}) \\ &= (\alpha [\downarrow]_F + [\downarrow]_G) \cdot X_{\downarrow} + \alpha \langle F \rangle_{-B} + \langle G \rangle_{-B} \\ &\quad + \psi(\alpha \llbracket P \rrbracket(\langle F \rangle_B) + \llbracket P \rrbracket(\langle G \rangle_B)) \quad (\text{Lin. of } \llbracket P \rrbracket) \\ &= (\alpha [\downarrow]_F + [\downarrow]_G) \cdot X_{\downarrow} + \alpha \langle F \rangle_{-B} + \langle G \rangle_{-B} \\ &\quad + \alpha \cdot \psi(\llbracket P \rrbracket(\langle F \rangle_B)) + \psi(\llbracket P \rrbracket(\langle G \rangle_B)) \quad (\text{Lin. of } \psi) \\ &= \alpha \cdot ([\downarrow]_F X_{\downarrow} + \langle F \rangle_{-B} + \psi(\llbracket P \rrbracket(\langle F \rangle_B))) \\ &\quad + ([\downarrow]_G X_{\downarrow} + \langle G \rangle_{-B} + \psi(\llbracket P \rrbracket(\langle G \rangle_B))) \\ &= \alpha \cdot \Phi_{B,P}(\psi)(F) + \Phi_{B,P}(\psi)(G) \end{aligned}$$

□

Corollary 38. *If $\llbracket P \rrbracket$ is linear, then $\Phi_{B,P}^n(\perp)$ is linear for all $n \in \mathbb{N}$, i.e.*

$$\Phi_{B,P}^n(\perp)(\alpha F + G) = \alpha \cdot \Phi_{B,P}^n(\perp)(F) + \Phi_{B,P}^n(\perp)(G).$$

PROOF. By induction:

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

Induction step: By the induction hypothesis $\Phi_{B,P}^n(\perp)$ is a linear function. Therefore, $\Phi_{B,P}^{n+1}(\perp) = \Phi_{B,P}(\Phi_{B,P}^n(\perp))$ is also linear by [Lemma 37](#). \square

Theorem 39 (Linearity of $\llbracket \cdot \rrbracket$). *The semantics transformer $\llbracket \cdot \rrbracket$ is linear, i.e. for any cpGCL program P*

$$\llbracket P \rrbracket(\alpha F + G) = \alpha \cdot \llbracket P \rrbracket(F) + \llbracket P \rrbracket(G).$$

PROOF. By induction over the structure of P :

Case $P = \text{skip}$:

$$\begin{aligned} \llbracket P \rrbracket(\alpha F + G) &= \alpha F + G \\ &= \alpha \llbracket P \rrbracket(F) + \llbracket P \rrbracket(G) \end{aligned}$$

Case $P = x_i := E$:

$$\begin{aligned} \llbracket P \rrbracket(\alpha F + G) &= \llbracket P \rrbracket((\alpha[\downarrow]_F + [\downarrow]_G)X_{\downarrow} \\ &\quad + \sum_{\sigma \in \mathbb{N}^k} (\alpha[\sigma]_F + [\sigma]_G)\mathbf{X}^\sigma) \\ &= (\alpha[\downarrow]_F + [\downarrow]_G)X_{\downarrow} \\ &\quad + \sum_{\sigma \in \mathbb{N}^k} (\alpha[\sigma]_F + [\sigma]_G)X_1^{\sigma_1} \dots X_i^{\text{eval}_\sigma(E)} \dots X_k^{\sigma_k} \\ &= \alpha \cdot \left([\downarrow]_F X_{\downarrow} + \sum_{\sigma \in \mathbb{N}^k} [\sigma]_F X_1^{\sigma_1} \dots X_i^{\text{eval}_\sigma(E)} \dots X_k^{\sigma_k} \right) \\ &\quad + \left([\downarrow]_G X_{\downarrow} + \sum_{\sigma \in \mathbb{N}^k} [\sigma]_G X_1^{\sigma_1} \dots X_i^{\text{eval}_\sigma(E)} \dots X_k^{\sigma_k} \right) \\ &= \alpha \cdot \llbracket P \rrbracket(F) + \llbracket P \rrbracket(G) \end{aligned}$$

Case $P = \text{observe } (B)$:

$$\begin{aligned} \llbracket P \rrbracket(\alpha F + G) &= (\alpha[\downarrow]_F + [\downarrow]_G + |\langle \alpha F + G \rangle_{-B}|)X_{\downarrow} + \langle \alpha F + G \rangle_B \\ &= (\alpha[\downarrow]_F + [\downarrow]_G + \alpha |\langle F \rangle_{-B}| + |\langle G \rangle_{-B}|)X_{\downarrow} \\ &\quad + \alpha \langle F \rangle_B + \langle G \rangle_B \quad (\text{Lin. of } \langle \cdot \rangle_B \text{ (Lemma 36)}) \\ &= \alpha (([\downarrow]_F + |\langle F \rangle_{-B}|)X_{\downarrow} + \langle F \rangle_B) \\ &\quad + (([\downarrow]_G + |\langle G \rangle_{-B}|)X_{\downarrow} + \langle G \rangle_B) \\ &= \alpha \llbracket P \rrbracket(F) + \llbracket P \rrbracket(G) \end{aligned}$$

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

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

Case $P = \text{if } (B) \{ P_1 \} \text{ else } \{ P_2 \}$:

$$\begin{aligned}
 & \llbracket P \rrbracket (\alpha F + G) \\
 &= (\alpha [\downarrow]_F + [\downarrow]_G) X_{\downarrow} + \llbracket P_1 \rrbracket (\langle \alpha F + G \rangle_B) + \llbracket P_2 \rrbracket (\langle \alpha F + G \rangle_{\neg B}) \\
 &= (\alpha [\downarrow]_F + [\downarrow]_G) X_{\downarrow} + \llbracket P_1 \rrbracket (\alpha \langle F \rangle_B + \langle G \rangle_B) \\
 &\quad + \llbracket P_2 \rrbracket (\alpha \langle F \rangle_{\neg B} + \langle G \rangle_{\neg B}) \quad (\text{Lin. of } \langle \cdot \rangle_B \text{ (Lemma 36)}) \\
 &= (\alpha [\downarrow]_F + [\downarrow]_G) X_{\downarrow} + \alpha \llbracket P_1 \rrbracket (\langle F \rangle_B) + \llbracket P_1 \rrbracket (\langle G \rangle_B) \\
 &\quad + \alpha \llbracket P_2 \rrbracket (\langle F \rangle_{\neg B}) + \llbracket P_2 \rrbracket (\langle G \rangle_{\neg B}) \quad (\text{I.H.}) \\
 &= \alpha ([\downarrow]_F X_{\downarrow} + \llbracket P_1 \rrbracket (\langle F \rangle_B) + \llbracket P_2 \rrbracket (\langle F \rangle_{\neg B})) \\
 &\quad + ([\downarrow]_G X_{\downarrow} + \llbracket P_1 \rrbracket (\langle G \rangle_B) + \llbracket P_2 \rrbracket (\langle G \rangle_{\neg B})) \\
 &= \alpha \llbracket P \rrbracket (F) + \llbracket P \rrbracket (G)
 \end{aligned}$$

Case $P = P_1 \circ P_2$:

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

Case $P = \text{while } (B) \{ P_1 \}$:

$$\begin{aligned}
 & \llbracket P \rrbracket (\alpha F + G) \\
 &= (\text{lfp } \Phi_{B, P_1})(\alpha F + G) \\
 &= \left(\sup \{ \Phi_{B, P_1}^n(\perp) \mid n \in \mathbb{N} \} \right) (\alpha F + G) \\
 &= \sup \{ \Phi_{B, P_1}^n(\perp)(\alpha F + G) \mid n \in \mathbb{N} \} \\
 &= \sup \{ \alpha \cdot \Phi_{B, P_1}^n(\perp)(F) + \Phi_{B, P_1}^n(\perp)(G) \mid n \in \mathbb{N} \} \quad (\text{Corollary 38, } \llbracket P_1 \rrbracket \text{ lin. by I.H.}) \\
 &= \alpha \cdot \sup \{ \Phi_{B, P_1}^n(\perp)(F) \mid n \in \mathbb{N} \} + \sup \{ \Phi_{B, P_1}^n(\perp)(G) \mid n \in \mathbb{N} \} \\
 &= \alpha \cdot \left(\sup \{ \Phi_{B, P_1}^n(\perp) \mid n \in \mathbb{N} \} \right) (F) + \left(\sup \{ \Phi_{B, P_1}^n(\perp) \mid n \in \mathbb{N} \} \right) (G) \\
 &= \alpha \cdot (\text{lfp } \Phi_{B, P_1})(F) + (\text{lfp } \Phi_{B, P_1})(G) \\
 &= \alpha \llbracket P \rrbracket (F) + \llbracket P \rrbracket (G)
 \end{aligned}$$

□

Lemma 40 (Error Term Pass-Through). *For every program P and every $F \in \text{eFPS}$, the error term $[\downarrow]_F X_\downarrow$ passes through the transformer unaffected, i.e.*

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

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

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

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

Case $P = \text{skip}$:

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

Case $P = x_i := E$:

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

Case $P = \text{observe } (B)$:

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

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

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

Case $P = \text{if } (B) \{ P_1 \} \text{ else } \{ P_2 \}$:

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

Case $P = P_1 \circ P_2$:

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

Case $P = \text{while } (B) \{ P_1 \}$:

We show that $\forall n \in \mathbb{N} : \Phi_{B,P_1}^{n+1}(\perp)(X_\sharp) = X_\sharp$:

$$\begin{aligned}
 \Phi_{B,P_1}^{n+1}(\perp)(X_\sharp) &= \Phi_{B,P_1}(\Phi_{B,P_1}^n(\perp))(X_\sharp) \\
 &= X_\sharp + \langle X_\sharp \rangle_{\neg B} + \Phi_{B,P_1}^n(\perp)(\llbracket P_1 \rrbracket(\langle X_\sharp \rangle_B)) && (\text{def. } \Phi_{B,P_1}) \\
 &= X_\sharp + \Phi_{B,P_1}^n(\perp)(0) \\
 &= X_\sharp && (\Phi_{B,P_1}^n(\perp)(0) \leq \llbracket \text{while } (B) \{ P_1 \} \rrbracket(0) = 0)
 \end{aligned}$$

From this, it follows:

$$\begin{aligned}
 \llbracket P \rrbracket(X_\sharp) &= \sup \{ \Phi_{B,P_1}^n(\perp)(X_\sharp) \mid n \in \mathbb{N} \} \\
 &= \sup \{ 0, X_\sharp \} && (\forall n \in \mathbb{N} : \Phi_{B,P_1}^{n+1}(\perp)(X_\sharp) = X_\sharp) \\
 &= X_\sharp
 \end{aligned}$$

□

Lemma 41 (Alternative Representation).

$$\begin{aligned}
 \llbracket \text{while } (B) \{ P \} \rrbracket(G) &= \sum_{i=0}^{\infty} \left([\sharp]_{\varphi_{B,P}^i(G)} X_\sharp + \langle \varphi_{B,P}^i(G) \rangle_{\neg B} \right) \\
 \text{where } \varphi_{B,P}(G) &:= \llbracket P \rrbracket(\langle G \rangle_B).
 \end{aligned}$$

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

$$\Phi_{B,P}^n(\perp)(G) = \sum_{i=0}^{n-1} \left([\sharp]_{\varphi_{B,P}^i(F)} X_\sharp + \langle \varphi_{B,P}^i(F) \rangle_{\neg B} \right).$$

Base case: $n = 0$.

$$\Phi_{B,P}^0(\perp)(G) = 0 = \sum_{i=0}^{-1} \left([\sharp]_{\varphi_{B,P}^i(G)} X_\sharp + \langle \varphi_{B,P}^i(G) \rangle_{\neg B} \right)$$

Induction step:

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

From this, it follows:

$$\begin{aligned}
\llbracket \text{while } (B) \{ P \} \rrbracket (G) &= \left(\sup_{n \in \mathbb{N}} \Phi_{B,P}^n(\perp) \right) (G) \\
&= \sup \{ \Phi_{B,P}^n(\perp)(G) \mid n \in \mathbb{N} \} \\
&= \sup_{n \in \mathbb{N}} \left\{ \sum_{i=0}^{n-1} [\downarrow]_{\varphi_{B,P}^i(G)} X_{\downarrow} + \langle \varphi_{B,P}^i(G) \rangle_{\neg B} \right\} \\
&= \sum_{i=0}^{\infty} [\downarrow]_{\varphi_{B,P}^i(G)} X_{\downarrow} + \langle \varphi_{B,P}^i(G) \rangle_{\neg B}
\end{aligned}$$

□

Lemma 42 (Infinite Applications of Linearity). *Let the linear and continuous function $\psi: \text{eFPS} \rightarrow \text{eFPS}$. Further, let $\alpha_i \in \mathbb{R}_{\geq 0}^{\infty}$ and $F_i \in \text{eFPS}$ 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.

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

□

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 $\mathcal{R}_{\sigma} \llbracket P \rrbracket$ precisely reflects the unconditioned PGF semantics $\llbracket P \rrbracket(\mathbf{X}^{\sigma})$ for any $p \in \text{cpGCL}$ with initial state $\sigma \in \mathbb{N}^k$. Lemma 44 shows that the probabilities $\Pr^{\mathcal{R}_{\sigma} \llbracket P \rrbracket}(\diamond \langle \downarrow, \sigma' \rangle)$ for all σ' and $\Pr^{\mathcal{R}_{\sigma} \llbracket P \rrbracket}(\diamond \downarrow)$ arising from the Markov chain correspond to the coefficients of $\llbracket P \rrbracket(\mathbf{X}^{\sigma})$. It is further shown that modifying these probabilities to the conditional probabilities $\Pr^{\mathcal{R}_{\sigma} \llbracket P \rrbracket}(\diamond \langle \downarrow, \sigma' \rangle \mid \neg \diamond \downarrow)$ has the same effect as applying the normalization function *norm*, thus concluding that the two semantics coincide (cf. Theorem 45).

Definition 43 (Markov Chain Semantics of cpGCL). *For any cpGCL program P and any starting state $\sigma \in \mathbb{N}^k$, the operational Markov chain is*

$$\mathcal{R}_{\sigma} \llbracket P \rrbracket \triangleq (\mathcal{S}, \langle P, \sigma \rangle, \mathcal{P}),$$

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. 4, then $s' \in \mathcal{S}$
- $\mathcal{P}: \mathcal{S} \times \mathcal{S} \rightarrow [0, 1]$ is the transition matrix with
 - $\mathcal{P}(s, s') = p$ if $s \xrightarrow{p} s'$ can be derived according to Fig. 4
 - $\mathcal{P}(s, s') = 0$ otherwise

$$\begin{array}{c}
\text{(skip)} \frac{}{\langle \text{skip}, \sigma \rangle \longrightarrow \langle \downarrow, \sigma \rangle} \quad \text{(asgn)} \frac{}{\langle x := E, \sigma \rangle \longrightarrow \langle \downarrow, \sigma[x \leftarrow \text{eval}_\sigma(E)] \rangle} \\
\text{(obs-t)} \frac{\sigma \models B}{\langle \text{observe } (B), \sigma \rangle \longrightarrow \langle \downarrow, \sigma \rangle} \quad \text{(obs-f)} \frac{\sigma \not\models B}{\langle \text{observe } (B), \sigma \rangle \longrightarrow \langle \downarrow, \sigma \rangle} \\
\text{(seq-1)} \frac{}{\langle \downarrow ; Q, \sigma \rangle \longrightarrow \langle Q, \sigma \rangle} \quad \text{(seq-2)} \frac{\langle P, \sigma \rangle \longrightarrow \langle \downarrow, \sigma \rangle}{\langle P ; Q, \sigma \rangle \longrightarrow \langle \downarrow, \sigma \rangle} \\
\text{(seq-3)} \frac{\langle P, \sigma \rangle \xrightarrow{p} \langle P', \sigma' \rangle}{\langle P ; Q, \sigma \rangle \xrightarrow{p} \langle P' ; Q, \sigma' \rangle} \\
\text{(choice-l)} \frac{}{\langle \{ P \} [p] \{ Q \}, \sigma \rangle \xrightarrow{p} \langle P, \sigma \rangle} \quad \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 \text{if } (B) \{ P \} \text{ else } \{ Q \}, \sigma \rangle \longrightarrow \langle P, \sigma \rangle} \quad \text{(if-f)} \frac{\sigma \not\models B}{\langle \text{if } (B) \{ P \} \text{ else } \{ Q \}, \sigma \rangle \longrightarrow \langle Q, \sigma \rangle} \\
\text{(while-t)} \frac{\sigma \models B}{\langle \text{while } (B) \{ P \}, \sigma \rangle \longrightarrow \langle P ; \text{while } (B) \{ P \}, \sigma \rangle} \quad \text{(while-f)} \frac{\sigma \not\models B}{\langle \text{while } (B) \{ P \}, \sigma \rangle \longrightarrow \langle \downarrow, \sigma \rangle} \\
\text{(terminal)} \frac{}{\langle \downarrow, \sigma \rangle \longrightarrow \langle \text{sink} \rangle} \quad \text{(undesired)} \frac{}{\langle \downarrow, \sigma \rangle \longrightarrow \langle \text{sink} \rangle} \quad \text{(sink)} \frac{}{\langle \text{sink} \rangle \longrightarrow \langle \text{sink} \rangle}
\end{array}$$

Fig. 4. Construction rules for the operational Markov chain. $\sigma[x \leftarrow \text{eval}_\sigma(E)]$ denotes the program state σ with the value of x replaced by $\text{eval}_\sigma(E)$. Whenever a transition has no annotated weight above its arrow, it has a weight of 1.

Lemma 44. For every $P \in \text{cpGCL}$ and every two $\sigma, \sigma' \in \mathbb{N}^k$

- (1) $\Pr^{\mathcal{R}_\sigma[P]}(\diamond \langle \downarrow, \sigma' \rangle) = [\sigma']_{\llbracket P \rrbracket(X^\sigma)}$
- (2) $\Pr^{\mathcal{R}_\sigma[P]}(\diamond \langle \downarrow, \sigma \rangle) = [\sigma]_{\llbracket P \rrbracket(X^\sigma)}$

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

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

$$\rightarrow \langle \text{skip}, \sigma \rangle \longrightarrow \langle \downarrow, \sigma \rangle \longrightarrow \langle \text{sink} \rangle \curvearrowright$$

$$\llbracket P \rrbracket(X^\sigma) = X^\sigma$$

Thus:

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

$$\rightarrow \langle x_i := E, \sigma \rangle \rightarrow \langle \downarrow, \sigma[x_i \leftarrow \text{eval}_\sigma(E)] \rangle \longrightarrow \langle \text{sink} \rangle \curvearrowright$$

Case $P = x_i := E$:

Its PGF semantics yields:

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

Thus:

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

$$\text{Pr}^{\mathcal{R}_\sigma \llbracket P \rrbracket}(\diamond \downarrow) = 0 = [\downarrow]_{\llbracket P \rrbracket(\mathbf{X}^\sigma)}$$

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

Observe passed:

$$\rightarrow \langle \text{observe } (B), \sigma \rangle \rightarrow \langle \downarrow, \sigma \rangle \longrightarrow \langle \text{sink} \rangle \curvearrowright$$

The PGF semantics yields:

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

Thus:

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

Observe failed:

$$\rightarrow \langle \text{observe } (B), \sigma \rangle \rightarrow \langle \downarrow \rangle \longrightarrow \langle \text{sink} \rangle \curvearrowright$$

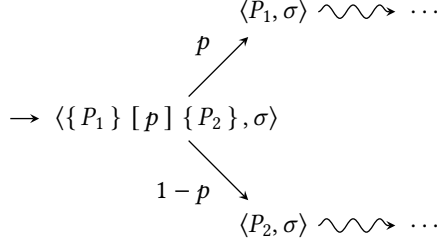
The PGF semantics yields:

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

Thus:

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

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



The PGF semantics yields:

$$\llbracket P \rrbracket(\mathbf{X}^\sigma) = p \cdot \llbracket P_1 \rrbracket(\mathbf{X}^\sigma) + (1-p) \cdot \llbracket P_2 \rrbracket(\mathbf{X}^\sigma)$$

Thus:

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

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

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

Condition is satisfied:

$$\rightarrow \langle \text{if } (B) \{ P_1 \} \text{ else } \{ P_2 \}, \sigma \rangle \triangleright \langle P_1, \sigma \rangle \rightsquigarrow \dots$$

The PGF semantics yields:

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

Thus:

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

$$\begin{aligned}
 \Pr^{\mathcal{R}_\sigma} \llbracket P \rrbracket(\diamond \langle \downarrow \rangle) &= [\downarrow]_{\llbracket P_1 \rrbracket(\mathbf{X}^\sigma)} \\
 &= [\downarrow]_{\llbracket P \rrbracket(\mathbf{X}^\sigma)}
 \end{aligned}$$

Condition not satisfied:

$$\rightarrow \langle \text{if } (B) \{ P_1 \} \text{ else } \{ P_2 \}, \sigma \rangle \triangleright \langle P_2, \sigma \rangle \rightsquigarrow \dots$$

The PGF semantics yields:

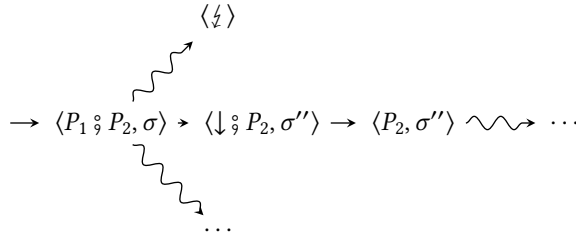
$$\llbracket P \rrbracket(\mathbf{X}^\sigma) = \llbracket P_1 \rrbracket(\langle \mathbf{X}^\sigma \rangle_B) + \llbracket P_2 \rrbracket(\langle \mathbf{X}^\sigma \rangle_{\neg B}) + [\downarrow]_{\mathbf{X}^\sigma} X_{\downarrow} = \llbracket P_2 \rrbracket(\mathbf{X}^\sigma)$$

Thus:

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

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

Case $P = P_1 \circ P_2$:



Case $P = \text{while}(B) \{ P_1 \}$:

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

$$\rightarrow \langle \text{while}(B) \{ P_1 \}, \sigma \rangle \rightarrow \langle \downarrow, \sigma \rangle \longrightarrow \langle \text{sink} \rangle \curvearrowright$$

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

$$\begin{aligned} \Phi_{B, P_1}^n(\perp)(\mathbf{X}^\sigma) &= [\downarrow]_{\mathbf{X}^\sigma} + \langle \mathbf{X}^\sigma \rangle_{\neg B} + \Phi_{B, P_1}^n(\perp)(\llbracket P_1 \rrbracket(\langle \mathbf{X}^\sigma \rangle_B)) \\ &= [\downarrow]_{\mathbf{X}^\sigma} + \Phi_{B, P_1}^n(\perp)(\llbracket P_1 \rrbracket(\langle \mathbf{X}^\sigma \rangle_B)) \\ &= 0 + \mathbf{X}^\sigma + 0 \\ \llbracket P \rrbracket(\mathbf{X}^\sigma) &= \text{lfp } \Phi_{B, P_1}(\mathbf{X}^\sigma) \\ &= \sup_{n \in \mathbb{N}} \{ \Phi_{B, P_1}^n(\perp)(\mathbf{X}^\sigma) \mid n \in \mathbb{N} \} \\ &= \mathbf{X}^\sigma \end{aligned} \quad (\sigma \not\models B)$$

Thus:

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

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

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

$$\begin{aligned} \Pr^{\mathcal{R}_\sigma}[P](\diamond \langle \downarrow, \sigma' \rangle) &= \sum_{n=1}^{\infty} \Pr^{\mathcal{R}_\sigma}[P](\diamond_{=n} \langle \downarrow, \sigma' \rangle) \\ \Pr^{\mathcal{R}_\sigma}[P](\diamond \langle \downarrow \rangle) &= \sum_{n=1}^{\infty} \Pr^{\mathcal{R}_\sigma}[P](\diamond_{=n} \langle \downarrow \rangle), \end{aligned}$$

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

By [Lemma 41](#), the PGF semantics can be represented as follows:

$$\begin{aligned} \llbracket P \rrbracket(\mathbf{X}^\sigma) &= \sum_{n=0}^{\infty} \left([\downarrow]_{\varphi_{B,P_1}^n(\mathbf{X}^\sigma)} X_{\downarrow} + \langle \varphi_{B,P_1}^n(\mathbf{X}^\sigma) \rangle_{-B} \right), \quad \text{where} \\ \varphi_{B,P_1}(F) &= \llbracket P_1 \rrbracket(\langle F \rangle_B). \end{aligned}$$

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

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

We can therefore restate the initial claims of [Lemma 44](#) as the following (stricter) conditions:

1. For all $n \in \mathbb{N}_{>0}$:

$$\Pr^{\mathcal{R}_\sigma}[P](\diamond_{=n} \langle \downarrow, \sigma' \rangle) = [\sigma']_{\langle \varphi_{B,P_1}^n(\mathbf{X}^\sigma) \rangle_{-B}}$$

2. For all $n \in \mathbb{N}_{>0}$:

$$\Pr^{\mathcal{R}_\sigma}[P](\diamond_{=n} \langle \downarrow \rangle) = [\downarrow]_{\varphi_{B,P_1}^n(\mathbf{X}^\sigma)}$$

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

$$\begin{aligned} \varphi_{B,P_1}^{n+1}(\mathbf{X}^\sigma) &= \varphi_{B,P_1}^n(\varphi_{B,P_1}(\mathbf{X}^\sigma)) \\ &= \varphi_{B,P_1}^n \left(\sum_{\sigma'' \in \mathbb{N}^k} [\sigma'']_{\llbracket P_1 \rrbracket(\mathbf{X}^\sigma)} \mathbf{X}^{\sigma''} \right) \\ &= \sum_{\sigma'' \in \mathbb{N}^k} [\sigma'']_{\llbracket P_1 \rrbracket(\mathbf{X}^\sigma)} \cdot \varphi_{B,P_1}^n(\mathbf{X}^{\sigma''}) \end{aligned} \tag{3}$$

1. First, note that a loop can never terminate in σ' if $\sigma' \models B$. Accordingly, the construction rules of the Markov chain semantics (cf. [Figure 4](#)) 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 $n \in \mathbb{N}_{>0}^k$):

$$\Pr^{\mathcal{R}_\sigma}[P](\diamond_{=n} \langle \downarrow, \sigma' \rangle) = [\sigma']_{\langle \varphi_{B,P_1}^n(\mathbf{X}^\sigma) \rangle_{-B}} = 0$$

We show the case $\sigma' \not\models B$ by induction:

Base case: $n = 1$.

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

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 σ'' , from which the loop then terminates in n iterations, i.e.:

$$\begin{aligned}
& \Pr^{\mathcal{R}_\sigma} \llbracket P \rrbracket (\diamond_{=n+1} \langle \downarrow, \sigma' \rangle) \\
&= \sum_{\sigma'' \in \mathbb{N}^k} \Pr^{\mathcal{R}_\sigma} \llbracket P_1 \rrbracket (\diamond \langle \downarrow, \sigma'' \rangle) \cdot \Pr^{\mathcal{R}_{\sigma''}} \llbracket P \rrbracket (\diamond_{=n} \langle \downarrow, \sigma' \rangle) \\
&= \sum_{\sigma'' \in \mathbb{N}^k} [\sigma''] \llbracket P_1 \rrbracket (X^\sigma) \cdot \Pr^{\mathcal{R}_{\sigma''}} \llbracket P \rrbracket (\diamond_{=n} \langle \downarrow, \sigma' \rangle) && \text{(outer I.H.)} \\
&= \sum_{\sigma'' \models B} [\sigma''] \llbracket P_1 \rrbracket (X^\sigma) \cdot \Pr^{\mathcal{R}_{\sigma''}} \llbracket P \rrbracket (\diamond_{=n} \langle \downarrow, \sigma' \rangle) && (0 \text{ if } \sigma'' \not\models B) \\
&= \sum_{\sigma'' \models B} [\sigma''] \llbracket P_1 \rrbracket (X^\sigma) \cdot [\sigma'] \langle \varphi_{B,P_1}^n (X^{\sigma''}) \rangle_{\neg B} && \text{(inner I.H.)} \\
&= \sum_{\sigma'' \in \mathbb{N}^k} [\sigma''] \llbracket P_1 \rrbracket (X^\sigma) \cdot [\sigma'] \langle \varphi_{B,P_1}^n (X^{\sigma''}) \rangle_{\neg B} && (0 \text{ if } \sigma'' \not\models B) \\
&= \sum_{\sigma'' \in \mathbb{N}^k} [\sigma'] \langle [\sigma''] \llbracket P_1 \rrbracket (X^\sigma) \cdot \varphi_{B,P_1}^n (X^{\sigma''}) \rangle_{\neg B} && \text{(Lin. of } [\sigma'] \text{ and } \langle \cdot \rangle_B) \\
&= [\sigma'] \langle \sum_{\sigma'' \in \mathbb{N}^k} [\sigma''] \llbracket P_1 \rrbracket (X^\sigma) \cdot \varphi_{B,P_1}^n (X^{\sigma''}) \rangle_{\neg B} && \text{(Lin. of } [\sigma'] \text{ and } \langle \cdot \rangle_B) \\
&= [\sigma'] \langle \varphi_{B,P_1}^{n+1} (X^\sigma) \rangle_{\neg B} && \text{(by Equation (3))}
\end{aligned}$$

2. By induction:

Base case: $n = 1$.

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

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

iteration, i.e.:

$$\begin{aligned}
& \Pr^{\mathcal{R}_\sigma} \llbracket P \rrbracket (\diamond_{=n+1} \langle \downarrow \rangle) \\
&= \sum_{\sigma'' \in \mathbb{N}^k} \Pr^{\mathcal{R}_\sigma} \llbracket P_1 \rrbracket (\diamond \langle \downarrow, \sigma'' \rangle) \cdot \Pr^{\mathcal{R}_{\sigma''}} \llbracket P \rrbracket (\diamond_{=n} \langle \downarrow \rangle) \\
&= \sum_{\sigma'' \in \mathbb{N}^k} [\sigma'']_{\llbracket P_1 \rrbracket (X^\sigma)} \cdot \Pr^{\mathcal{R}_{\sigma''}} \llbracket P \rrbracket (\diamond_{=n} \langle \downarrow \rangle) && \text{(outer I.H.)} \\
&= \sum_{\sigma'' \models B} [\sigma'']_{\llbracket P_1 \rrbracket (X^\sigma)} \cdot \Pr^{\mathcal{R}_{\sigma''}} \llbracket P \rrbracket (\diamond_{=n} \langle \downarrow \rangle) && (0 \text{ if } \sigma'' \not\models B) \\
&= \sum_{\sigma'' \models B} [\sigma'']_{\llbracket P_1 \rrbracket (X^\sigma)} \cdot [\downarrow]_{\varphi_{B,P_1}^n (X^{\sigma''})} && \text{(inner I.H.)} \\
&= \sum_{\sigma'' \in \mathbb{N}^k} [\sigma'']_{\llbracket P_1 \rrbracket (X^\sigma)} \cdot [\downarrow]_{\varphi_{B,P_1}^n (X^{\sigma''})} && (0 \text{ if } \sigma'' \not\models B) \\
&= \sum_{\sigma'' \in \mathbb{N}^k} [\downarrow]_{[\sigma'']_{\llbracket P_1 \rrbracket (X^\sigma)} \cdot \varphi_{B,P_1}^n (X^{\sigma''})} && \text{(Lin. of } [\downarrow] \text{)} \\
&= [\downarrow]_{\sum_{\sigma'' \in \mathbb{N}^k} [\sigma'']_{\llbracket P_1 \rrbracket (X^\sigma)} \cdot \varphi_{B,P_1}^n (X^{\sigma''})} && \text{(Lin. of } [\downarrow] \text{ and } \langle \cdot \rangle_B \text{)} \\
&= [\downarrow]_{\varphi_{B,P_1}^{n+1} (X^\sigma)} && \text{(by Equation (3))}
\end{aligned}$$

□

Theorem 45 (Operational Equivalence). For every cpGCL program p and every $\sigma, \sigma' \in \mathbb{N}^k$

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

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.

$$\begin{aligned}
\Pr^{\mathcal{R}_\sigma} \llbracket P \rrbracket (\diamond \langle \downarrow, \sigma' \rangle \mid \neg \diamond \downarrow) &= \frac{\Pr^{\mathcal{R}_\sigma} \llbracket P \rrbracket (\diamond \langle \downarrow, \sigma' \rangle \wedge \neg \diamond \downarrow)}{\Pr^{\mathcal{R}_\sigma} \llbracket P \rrbracket (\neg \diamond \downarrow)} \\
&= \frac{\Pr^{\mathcal{R}_\sigma} \llbracket P \rrbracket (\diamond \langle \downarrow, \sigma' \rangle)}{\Pr^{\mathcal{R}_\sigma} \llbracket P \rrbracket (\neg \diamond \downarrow)} && \text{(reaching } \langle \downarrow, \sigma' \rangle \text{ implies not reaching } \langle \downarrow \rangle) \\
&= \frac{\Pr^{\mathcal{R}_\sigma} \llbracket P \rrbracket (\diamond \langle \downarrow, \sigma' \rangle)}{1 - \Pr^{\mathcal{R}_\sigma} \llbracket P \rrbracket (\diamond \downarrow)} \\
&= \frac{[\sigma']_{\llbracket P \rrbracket (X^\sigma)}}{1 - [\downarrow]_{\llbracket P \rrbracket (X^\sigma)}} && \text{(cf. Lemma 44)} \\
&= [\sigma']_{\text{norm}(\llbracket P \rrbracket (X^\sigma))}
\end{aligned}$$

□

C REASONING ABOUT LOOPS

Definition 46 (Admissible eSOP-transformer). A function $\psi: \text{eSOP} \rightarrow \text{eSOP}$ is called admissible if

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

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

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

$$\psi(GU^\tau) = \psi(G)U^\tau.$$

- ψ preserves ePGF, i.e., $G \in \text{ePGF}$ implies $\psi(G) \in \text{ePGF}$.

Theorem 47 (SOP Semantics). *Let P be a loop-free cReDiP Program. Let $G = \sum_{\sigma \in \mathbb{N}^k} G_\sigma U^\sigma \in \text{eSOP}$. The eSOP semantics of P is then given by:*

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

PROOF OUTLINE. The proof of [Theorem 13](#) proceeds along a similar line of reasoning as in [\[Chen et al. 2022\]](#). First of all, we use [Lemma 42](#) and obtain:

$$\begin{aligned} \llbracket P \rrbracket(G) &= \sum_{\sigma \in \mathbb{N}^k} \llbracket P \rrbracket(G_\sigma U^\sigma) \\ &= \sum_{\sigma \in \mathbb{N}^k} \llbracket P \rrbracket \left(\sum_{\tau \in \mathbb{N}^k} [\tau]_{G_\sigma} X^\tau U^\sigma + [\downarrow]_{G_\sigma} X_\downarrow U^\sigma \right) && \text{(by eFPS arithmetic)} \\ &= \sum_{\sigma \in \mathbb{N}^k} \sum_{\tau \in \mathbb{N}^k} \llbracket P \rrbracket ([\tau]_{G_\sigma} X^\tau U^\sigma + [\downarrow]_{G_\sigma} X_\downarrow U^\sigma) && \text{(by Lemma 42)} \\ &= \sum_{\sigma \in \mathbb{N}^k} \sum_{\tau \in \mathbb{N}^k} \llbracket P \rrbracket ([\tau]_{G_\sigma} X^\tau U^\sigma) + \llbracket P \rrbracket ([\downarrow]_{G_\sigma} X_\downarrow U^\sigma) && \text{(by Theorem 39)} \\ &= \sum_{\sigma \in \mathbb{N}^k} \sum_{\tau \in \mathbb{N}^k} [\tau]_{G_\sigma} \cdot \llbracket P \rrbracket (X^\tau U^\sigma) + [\downarrow]_{G_\sigma} X_\downarrow U^\sigma && \text{(by Lemma 12, Theorem 39)} \end{aligned}$$

Thus we need to show, that $\llbracket P \rrbracket(X^\tau U^\sigma)$ is homogeneous for all $\tau, \sigma \in \mathbb{N}$. All loop-free cases but observe coincide with ReDiP [\[Chen et al. 2022\]](#) on the distributions where the observe violation probability is zero which is an immediate consequence of [Lemma 12](#) 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: $G[X/1, X_\downarrow/1] \cdot X_\downarrow$. Note that the observe (false) semantics is entirely based on the following elementary transformations, which are admissible (by [\[Chen et al. 2022\]](#)):

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

Thus, $\llbracket \text{observe (false)} \rrbracket (X^\tau U^\sigma) = \llbracket \text{observe (false)} \rrbracket (X^\tau) U^\sigma$ 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 $\llbracket \text{observe (false)} \rrbracket (X^\tau U^\sigma) = 1^\tau X_\downarrow U^\sigma = \llbracket \text{observe (false)} \rrbracket (X^\tau) U^\sigma$. Combining that with

the equation from above, we conclude

$$\begin{aligned}
\sum_{\sigma \in \mathbb{N}^k} \sum_{\tau \in \mathbb{N}^k} [\tau]_{G_\sigma} \cdot \llbracket P \rrbracket (\mathbf{X}^\tau \mathbf{U}^\sigma) + [\downarrow]_{G_\sigma} X_{\downarrow} \mathbf{U}^\sigma &= \sum_{\sigma \in \mathbb{N}^k} \sum_{\tau \in \mathbb{N}^k} [\tau]_{G_\sigma} \cdot \llbracket P \rrbracket (\mathbf{X}^\tau) \mathbf{U}^\sigma + [\downarrow]_{G_\sigma} X_{\downarrow} \mathbf{U}^\sigma \\
&= \sum_{\sigma \in \mathbb{N}^k} \sum_{\tau \in \mathbb{N}^k} \llbracket P \rrbracket ([\tau]_{G_\sigma} \cdot \mathbf{X}^\tau) \mathbf{U}^\sigma + \llbracket P \rrbracket ([\downarrow]_{G_\sigma} X_{\downarrow}) \mathbf{U}^\sigma \\
&\quad \text{(by Theorem 39)} \\
&= \sum_{\sigma \in \mathbb{N}^k} \sum_{\tau \in \mathbb{N}^k} (\llbracket P \rrbracket ([\tau]_{G_\sigma} \cdot \mathbf{X}^\tau) + \llbracket P \rrbracket ([\downarrow]_{G_\sigma} X_{\downarrow})) \mathbf{U}^\sigma \\
&\quad \text{(by eFPS arithmetic)} \\
&= \sum_{\sigma \in \mathbb{N}^k} \sum_{\tau \in \mathbb{N}^k} (\llbracket P \rrbracket ([\tau]_{G_\sigma} \cdot \mathbf{X}^\tau + [\downarrow]_{G_\sigma} X_{\downarrow})) \mathbf{U}^\sigma \\
&\quad \text{(Theorem 39)} \\
&= \sum_{\sigma \in \mathbb{N}^k} \llbracket P \rrbracket (G_\sigma) \mathbf{U}^\sigma \quad \text{(by Def. of } G_\sigma)
\end{aligned}$$

□

D PARAMETER SYNTHESIS

Theorem 48 (Decidability of Parameter Synthesis). *Let W be a cReDiP while loop and Q_p be a loop-free cReDiP program. It is decidable whether there exist parameter values \mathbf{p} such that the instantiated template Q_p is an inductive invariant, i.e., $\exists \mathbf{p} \in \mathbb{R}^l. \forall \mathbf{X} \in \text{Vars}(W). \llbracket W \rrbracket = \llbracket Q_p \rrbracket$.*

PROOF. Let W and Q_p be given as described. Also, let $g = (1 - X_1 U_1)^{-1} \cdots (1 - X_k U_k)^{-1} \in \text{SOP}$.

$$\begin{aligned}
&\exists \mathbf{p} \in \mathbb{R}^l. \forall \mathbf{X}. \quad \llbracket Q_p \rrbracket = \llbracket \Phi_{B,P}(Q_p) \rrbracket \\
&\Leftrightarrow \exists \mathbf{p} \in \mathbb{R}^l. \forall \mathbf{X}. \forall \mathbf{U}. \quad \llbracket Q_p \rrbracket (g) = \llbracket \Phi_{B,P}(Q_p) \rrbracket (g) \quad \text{(Lemma 14)} \\
&\Leftrightarrow \exists \mathbf{p} \in \mathbb{R}^l. \forall \mathbf{X}. \forall \mathbf{U}. \quad \frac{F(\mathbf{X}, X_{\downarrow}, \mathbf{U}, \mathbf{p})}{H(\mathbf{X}, X_{\downarrow}, \mathbf{U}, \mathbf{p})} = \frac{\hat{F}(\mathbf{X}, X_{\downarrow}, \mathbf{U}, \mathbf{p})}{\hat{H}(\mathbf{X}, X_{\downarrow}, \mathbf{U}, \mathbf{p})} \\
&\quad \text{(loop-free cReDiP preserves rational functions)} \\
&\Leftrightarrow \exists \mathbf{p} \in \mathbb{R}^l. \forall \mathbf{X}. \forall \mathbf{U}. \quad F(\mathbf{X}, X_{\downarrow}, \mathbf{U}, \mathbf{p}) \hat{H}(\mathbf{X}, X_{\downarrow}, \mathbf{U}, \mathbf{p}) = \hat{F}(\mathbf{X}, X_{\downarrow}, \mathbf{U}, \mathbf{p}) H(\mathbf{X}, X_{\downarrow}, \mathbf{U}, \mathbf{p}) \\
&\Leftrightarrow \exists \mathbf{p} \in \mathbb{R}^l. \forall \mathbf{X}. \forall \mathbf{U}. \quad F(\mathbf{X}, \mathbf{U}, \mathbf{p}) \hat{H}(\mathbf{X}, \mathbf{U}, \mathbf{p}) - \hat{F}(\mathbf{X}, \mathbf{U}, \mathbf{p}) H(\mathbf{X}, \mathbf{U}, \mathbf{p}) = 0
\end{aligned}$$

In the last step $F(\mathbf{X}, X_{\downarrow}, \mathbf{U}, \mathbf{p}) \hat{H}(\mathbf{X}, X_{\downarrow}, \mathbf{U}, \mathbf{p})$ and $\hat{F}(\mathbf{X}, X_{\downarrow}, \mathbf{U}, \mathbf{p}) H(\mathbf{X}, X_{\downarrow}, \mathbf{U}, \mathbf{p})$ are polynomials in $\mathbb{R}[\mathbf{p}][\mathbf{X}, X_{\downarrow}, \mathbf{U}]$. 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 $O\left(2^{2^{|\mathbf{X}|+|\mathbf{U}|+1}}\right)$ worst-case complexity to decide whether the formula can be satisfied. □

E ADDITIONAL EXAMPLES

Example 49. This example of two programs I and J shows the step-by-step computation of the invariant and the modified invariant to proof the actual equivalence.

$$\begin{aligned}
 I : \quad & \llbracket (1 - XU)^{-1}(1 - YV)^{-1} \\
 & \text{if } (y = 1) \{ \\
 & \quad \llbracket (1 - XU)^{-1}YV \\
 & \quad x += \text{iid}(\text{geom}(1/2) + 1, y) \\
 & \quad \llbracket (1 - XU)^{-1}X(2 - X)^{-1}YV \\
 & \quad y := 0 \\
 & \quad \llbracket (1 - XU)^{-1}X(2 - X)^{-1}V \\
 & \quad \text{observe } (x < 3) \\
 & \quad \llbracket (1/2X + 1/4X^2 + 1/4X_{\frac{1}{2}})V \\
 & \quad \quad + ((1/2X^2 + 1/2X_{\frac{1}{2}})U + X_{\frac{1}{2}}U^2(1 - U)^{-1})V \\
 & \quad \} \\
 & \llbracket (1 - XU)^{-1}(1 - YV)^{-1} - (1 - XU)^{-1}YV \\
 & \quad + (1/2X + 1/4X^2 + 1/4X_{\frac{1}{2}})V \\
 & \quad + ((1/2X^2 + 1/2X_{\frac{1}{2}})U + X_{\frac{1}{2}}U^2(1 - U)^{-1})V
 \end{aligned}$$

We want to show that $\llbracket J \rrbracket(\hat{G})$ — where $J = \text{if } (y = 1) \{P \ ; \ I\} \text{ else } \{\text{skip}\}$ — yields the same result:

$$\begin{aligned}
J : & \quad \llbracket (1 - XU)^{-1}(1 - YV)^{-1} \\
& \quad \text{if } (y = 1) \{ \\
& \quad \quad \llbracket (1 - XU)^{-1}YV \\
& \quad \quad \{ y := 0 \} [1/2] \{ y := 1 \} ; \\
& \quad \quad \llbracket 1/2(1 - XU)^{-1}(Y + 1)V \\
& \quad \quad x := x + 1 ; \\
& \quad \quad \llbracket 1/2X(1 - XU)^{-1}(Y + 1)V \\
& \quad \quad \text{observe } (x < 3) ; \\
& \quad \quad \llbracket (1/2(X + X^2U)(Y + 1) + X_{\frac{1}{2}}U^2(1 - U)^{-1})V \\
& \quad \quad \text{if } (y = 1) \{ \\
& \quad \quad \quad \llbracket 1/2(X + X^2U)YV \\
& \quad \quad \quad x += \text{iid}(\text{geom}(1/2) + 1, y) ; \\
& \quad \quad \quad \llbracket 1/2(X^2 + X^3U)(2 - X)^{-1}YV \\
& \quad \quad \quad y := 0 ; \\
& \quad \quad \quad \llbracket 1/2(X^2 + X^3U)(2 - X)^{-1}V \\
& \quad \quad \quad \text{observe } (x < 3) \\
& \quad \quad \quad \llbracket 1/2(1/2X^2 + 1/2X_{\frac{1}{2}} + X_{\frac{1}{2}}U)V \\
& \quad \quad \} \\
& \quad \quad \llbracket ((1/2X + 1/4X^2 + 1/4X_{\frac{1}{2}}) + (1/2X^2 + 1/2X_{\frac{1}{2}})U)V \\
& \quad \quad \quad + X_{\frac{1}{2}}U^2(1 - U)^{-1}V \\
& \quad \} \\
& \quad \llbracket (1 - XU)^{-1}(1 - YV)^{-1} - (1 - XU)^{-1}YV \\
& \quad \quad + ((1/2X + 1/4X^2 + 1/4X_{\frac{1}{2}}) + (1/2X^2 + 1/2X_{\frac{1}{2}})U)V \\
& \quad \quad + X_{\frac{1}{2}}U^2(1 - U)^{-1}V
\end{aligned}$$