Final Report: Verifying Probabilistic Program Equivalence via PGF Transformer Semantics

Cheng Peng (2020533068)

Finished on June 14, 2024

Abstract

Randomness is ubiquitous in the real world, and introducing randomness is inevitable in various domains, such as machine learning and cryptography, giving rise to probabilistic programming. While probabilistic programming is widely applied in security-critical applications, e.g., online banking and cyber-physical systems, there is a lack of effort in ensuring their correctness.

Recently, researchers proposed generating function transformer semantics to address the insufficiency of sensitivity and expressiveness of existing techniques. This novel denotational semantics allows tracking the entire distribution of program states, enabling precise reasoning about the behavior of probabilistic programs.

In this course project, we investigate the PGF transformer semantics of the ReDiP programming language and develop a proof-of-concept tool for equivalence checking.

1 Introduction

1.1 Motivation

Randomness is pervasive, arising naturally and inevitably in many contexts. For example, applications interacting with the physical world must handle noisy inputs, molecular dynamics involve stochastic motions, and random tie-breaking is crucial for efficient approximate algorithms. In recent years, there has been a surge in probabilistic programming, which unifies statistical modeling with general-purpose programming, making it much easier to program with randomness.

Probabilistic programming is increasingly employed in safety- and security-critical applications, such as healthcare, financial services, and autonomous control systems, where faults and vulnerabilities may lead to severe damage and significant costs. Therefore, formally verifying probabilistic programs is essential to ensure their reliability and build trust in these systems.

While the need for verifying probabilistic programs is ever-growing, verification techniques are still in an early stage. Most existing approaches focus on computing moments of variables[10], deriving probability of assertion violations[11], or establishing lower/upper bounds[1]. Though they are capable of detecting bugs and side-channel leaks in programs, they generally cannot determine the entire output distribution of programs, making them insensitive to tiny deviations from the desired distribution and unable to verify complex properties such as relations between variables. The limitations of previous works motivate people to seek a proper theoretical foundation for precise and versatile automated reasoning of probabilistic programs.

1.2 Overview

This project aims to develop a verifier to determine whether two probabilistic programs generate exactly the same output distribution for every possible input. Given that the equivalence checking problem for pGCL is undecidable, we restrict the probabilistic programming language to make equivalence decidable. The two main restrictions we introduce are (1) arithmetic expressions must be linear, and (2) comparisons must be rectangular. The resulting fragment is called rectangular discrete probabilistic programming language (ReDiP), a constrained yet expressive language. It turns out that equivalence checking is decidable for ReDiP.

To precisely reason about probabilistic programs, formal semantics have to be devised. Inspired by Kozen[6], Chen et al.[8] developed a denotational semantics based on probability generating functions

(PGFs) to capture the evolution of the distribution of program states as the program executes. We investigate the PGF transformer semantics, demonstrating the recursive calculation of the PGF transform of a ReDiP program. We also prove linearity and rational closed-form preservation of the PGF transformer semantics. Leveraging the linearity of PGF transformers of ReDiP programs, we re-implement the PRODIGY equivalence checker in Python. Finally, we use a set of tests from PRODIGY[2] to benchmark the performance and scalability of our tool.

The main work done in this project are:

- We define the ReDiP programming language, a restricted fragment of pGCL amenable to automated reasoning.
- We define the PGF transformer semantics and demonstrate the forward reasoning of probabilistic programs leveraging the PGF transform semantics.
- We establish properties of PGF transformers of ReDiP programs and implement an equivalence verifier leveraging those properties.
- We benchmark our tool to show its effectiveness and discover a potential scalability issue.

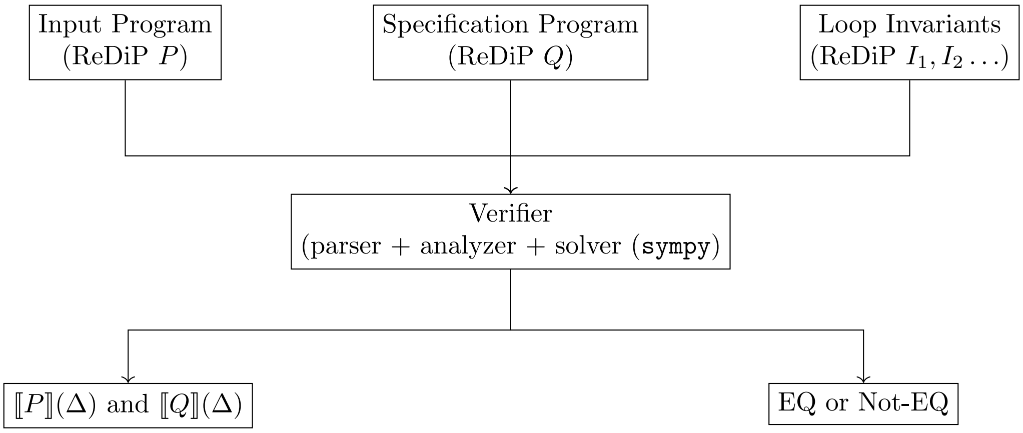
Input Program
(ReDiP P)
Specification Program
(ReDiP Q)
Loop Invariants
(ReDiP $I_1, I_2 \ldots$)
Verifier
(parser + analyzer + solver (`sympy`)
$[\![P]\!](\Delta)$ and $[\![Q]\!](\Delta)$
EQ or Not-EQ

Figure 1: Architecture of the verifier

2 Probabilistic Programming Languages

In this section, we introduce the syntax and semantics of the pGCL probabilistic programming language. We later impose several reasonable constraints on pGCL to obtain a restricted yet expressive language, ReDiP.

2.1 The pGCL programming language

The probabilistic guarded command language[3] is a simple imperative programming language augmented with probabilistic choice and random sampling. The syntax 2.1 describes what constitutes a valid pGCL program, and the semantics 2.2 defines the step-by-step execution of a pGCL program.

Definition 2.1 (pGCL syntax constructions)**.** *A pGCL program is a terminal command, an assignment command, or a compositional command, where:*

terminal command *A terminal command is a* skip *command.*

assignment command *Arithmetic expression $x := E$. Random sampling $x :\approx \mu$.*

compositional commands *Suppose that C_1, C_2 are valid pGCL programs, p is an arithmetic expression, and φ is a conditional (boolean) expression, then*

- *$\{C_1\}[p]\{C_2\}$ is a probabilistic choice command*
- *$C_1; C_2$ is a sequencing command*

- $\mathrm{if}(\varphi)\{C_1\}\,\mathrm{else}\{C_2\}$ *is a braching command*
- $\mathrm{while}(\varphi)\{C_1\}$ *is a looping command*

Definition 2.2 (pGCL operational semantics)**.** *The semantics of a pGCL program is defined as:*

- *A configuration space*

$$\mathcal{K} = \{\langle C, \sigma, n, \theta, q\rangle \mid C \in p\,\mathrm{GCL} \cup \{\downarrow\}, \sigma : \Sigma \to \mathbb{Z}, n \in \mathbb{N}, \theta \in \{L, R\}^*, q \in [0, 1]\}$$

  - *C is the pGCL command to be executed or a $\downarrow$ representing halt without error.*
  - *σ is the variable valuation function.*
  - *n is the number of steps up to now.*
  - *θ records all probabilistic choices made in the past.*
  - *q is the probability of reaching such a configuration.*

- *A transition relation $\vdash \subseteq \mathcal{K} \times \mathcal{K}$, defined by a set of rules:*
  - *skip:* $\langle \mathrm{skip}, \sigma, n, \theta, q\rangle \vdash \langle \downarrow, \sigma, n+1, \theta, q\rangle$
  - *arithmetic: if $\sigma(E) = a$ then* $\langle x := E, \sigma, n, \theta, q\rangle \vdash \langle \downarrow, \sigma[x \mapsto a], n+1, \theta, q\rangle$
  - *sampling: if $\mu(a) = r > 0$ then* $\langle x :\approx \mu, \sigma, n, \theta, q\rangle \vdash \langle \downarrow, \sigma[x \mapsto a], n+1, \theta, qr\rangle$
  - *probabilistic choice: If $\sigma(p) = r > 0$ then* $\langle \{C_1\}[p]\{C_2\}, \sigma, n, \theta, q\rangle \vdash \langle C_1, \sigma, n+1, \theta \cdot L, qr\rangle$
    If $1 - \sigma(p) = r' > 0$ then $\langle \{C_1\}[p]\{C_2\}, \sigma, n, \theta, q\rangle \vdash \langle C_2, \sigma, n+1, \theta \cdot R, qr'\rangle$
  - *sequencing: If* $\langle C_1, \sigma, n, \theta, q\rangle \vdash \langle C_1', \sigma', n', \theta', q'\rangle$ *then* $\langle C_1; C_2, \sigma, n, \theta, q\rangle \vdash \langle C_1'; C_2, \sigma', n', \theta', q'\rangle$
    Also $\langle \downarrow; C_2, \sigma, n, \theta, q\rangle \vdash \langle C_2, \sigma', n', \theta', q'\rangle$
  - *branching: If $\sigma(\varphi) = \mathbf{T}$ then* $\langle \mathrm{if}(\varphi)\{C_1\}\,\mathrm{else}\{C_2\} = \sigma, n, \theta, q\rangle \vdash \langle C_1, \sigma, n+1, \theta, q\rangle$*.*
    If $\sigma(\varphi) = \mathbf{F}$ then $\langle \mathrm{if}(\varphi)\{P\}\,\mathrm{else}\{Q\} = \sigma, n, \theta, q\rangle \vdash \langle C_2, \sigma, n+1, \theta, q\rangle$*.*
  - *looping: If $\sigma(\varphi) = \mathbf{T}$ then* $\langle \mathrm{while}(\varphi)\{C\}, \sigma, n, \theta, q\rangle \vdash \langle C; \mathrm{while}(\varphi)\{C\}, \sigma, n+1, \theta, q\rangle$*.*
    If $\sigma(\varphi) = \mathbf{F}$ then $\langle \mathrm{while}(\varphi)\{C\}, \sigma, n, \theta, q\rangle \vdash \langle \downarrow, \sigma, n+1, \theta, q\rangle$*.*

  In the above paragraph, $\sigma(E)$ and $\sigma(\varphi)$ are the value of arithmetic expression E and the value of conditional expression under valuation σ, respectively.

- *The (possibly infinite) computation tree of program C on initial variable valuation σ, where $k_0 = \langle C, \sigma, 0, \epsilon, 1\rangle$ is the root configuration. The tree contains all reachable configurations. Each configuration k except for the root is linked to the configuration k' where $k' \vdash k$.*

We now define the terminal configurations 2.3 of pGCL programs and formalize the intuition of equivalent programs 2.4.

Definition 2.3 (Terminal configurations)**.** *Terminal configuration for program C on input σ_0 is the set of all reachable halting configurations*

$$T(C, \sigma_0) = \{\langle \downarrow, \sigma, n, \theta, q\rangle \mid \langle C, \sigma_0, 0, \epsilon, 1\rangle \vdash^* \langle \downarrow, \sigma, n, \theta, q\rangle\}$$

The terminal valuation is defined as all possible valuations in terminal configurations.

$$TV(C, \sigma_0) = \{\sigma \mid \langle \downarrow, \sigma, n, \theta, q\rangle \in T(C, \sigma_0)\}$$

Similarly, terminal distribution is defined as the probability distribution of terminal valuations.

$$TD(C, \sigma_0; \sigma) = \sum_{\langle \downarrow, \sigma, n, \theta, q\rangle \in TV(C, \sigma_0)} q$$

Definition 2.4 (pGCL output equivalence)**.** *Two pGCL programs C_1, C_2 are said to be equivalent if for every input σ the terminal distribution is identical.*

$$\forall \sigma_0 . \forall \sigma . TD(C_1, \sigma_0; \sigma) \equiv TD(C_2, \sigma_0; \sigma)$$

2.2 The Linear and Rectangular Fragment of pGCL, ReDiP

The pGCL probabilistic programming language can encode arbitrary probabilistic Turing machine. Rice theorem suggests that there is no general sound and complete algorithm for analyzing pGCL programs. With that said, there are fragments of pGCL on which automated analysis are tractable. ReDiP is the fragment we will be working on.

Definition 2.5 (ReDiP programming language)**.** *A ReDiP program C is a pGCL program that*

- *Variables in C only takes natural number values.*
- *Arithmetic expressions E in $x := E$ are linear. They take the form $E = y + n$ where $y, n \in \{-1, 0, 1\} \cup \Sigma$.*
- *Conditional expressions φ in* $\text{if}(\varphi)\{P\}\,\text{else}\{Q\}$ *and* $\text{while}(\varphi)\{P\}$ *are rectangular: They take the form $\varphi = x < n$ where $x \in \Sigma$ and $n \in \mathbb{Z}$.*

To further enrich the fragment, introduce the following syntactic sugars:

- Linear combination expression command

$$x := a_0 + a_1x_1 + a_2x_2 \cdots a_nx_n \quad \text{where } a_i \in \mathbb{Z}, x_i \in \Sigma$$

  A linear combination expression command can be expressed as a sequential combination of finite linear expressions. For example $x := y + 2z - 3$ is equivalent to

$$\begin{aligned}
x &:= y + 0;\\
x &:= x + z;\\
x &:= x + z;\\
x &:= x - 1;\\
x &:= x - 1;\\
x &:= x - 1
\end{aligned}$$

- Finite loop command $\text{loop}(n)\{C\}$. The command is equivalent to $\overbrace{C; C; \cdots; C}^{n \text{ copies}}$.
- Logical connectives in conditional expressions: conjunction $\varphi \wedge \varphi$, disjunction $\varphi \vee \varphi$ and negation $\neg\varphi$.
- Extended rectangular conditions: $x\langle op\rangle n$ where $\langle op\rangle \in \{=, \neq, <, >, \leq, \geq\}$.
- Odd-Even conditional expression: $x \bmod 2 = 0$ and $x \bmod 2 = 1$.
- IID sampling command $x := \text{iid}(D, y)$ where D is a discrete probability distribution and x, y are variables.

We note that ReDiP is rich enough to write many practical programs, e.g., random walk simulation, simple rejection sampling, and Discrete Markov chains.

3 Review on GFs and PGFs

We first define the following notations that will be used frequently in this section:

- Vectors $\mathbf{x} = (x_1, x_2, \ldots x_k)$ and $\sigma : \{1, \ldots, k\} \mapsto \mathbb{N}$
- Monomials $\mathbf{x}^\sigma = x_1^{\sigma(1)}x_2^{\sigma(2)}\ldots x_k^{\sigma(k)} = \prod_i x_i^{\sigma(i)}$
- Formal power series (or polynomials): $P(\mathbf{x}) = \sum_\sigma f(\sigma)\mathbf{x}^\sigma$ where $f : \mathbb{N}^k \to \mathbb{R}$

3.1 Generating Functions

Generating functions[12] (GFs) are power tools originating from the field of enumerative combinatorics. GFs are also known as Z-transforms in the context of signal processing. The GF of a sequence is a formal power series whose coefficients are the elements of the sequence. For example the generating function of $\{a_n\}$ with meta-variable x is

$$G(a_n; x) = \sum_{n=0}^{\infty} a_n x^n$$

Manipulations on sequence can be mapped to algebraic operations on GFs.

Theorem 3.1 (GF operations for sequence manipulations)**.** *Suppose that the GF of $\{a_n\}$ and $\{b_n\}$ are $f = G(a_n, t)$ and $g = G(b_n, t)$ respectively, then*

$$\begin{aligned} G\left(a_{n-k}; x\right) &= x^k f \\ G\left(a_n + b_n; x\right) &= f + g \\ G\left(ra_n; x\right) &= rf \\ G\left(\sum_{k=0}^{n} a_k b_{n-k}; x\right) &= fg \\ G\left(na_n; x\right) &= x\partial_x f \\ G\left(a_{n-1}/n; x\right) &= \int f \mathrm{d}x \end{aligned}$$

Generating function $g(x)$	sequence $[x^n]g(x)$	note
$(1-ax)^k$	$\binom{k}{n}a^n$	$a, k \in \mathbb{R}^*$
$(1-x)^{-1}$	1	
$(1-x)^{-2}$	$n+1$	
x^k	$[n=k]$	$k \in \mathbb{N}^+$
e^{kx}	$k^n/n!$	$k \in \mathbb{R}^*$
$-\ln(1-x)$	$1/n$	$n \geq 1$

Table 1: Elementary generating functions and their corresponding sequences

Certain formal power series have equivalent elementary closed-form representations; they can be regarded as finite representations of infinite sequences. Table 3.1 summarizes typical closed-form GFs. Working on closed-form generation functions of sequences is easy since elementary functions are closed under arithmetics, composition, and differentiation. Furthermore, indefinite integration of elementary functions can be computed via the Risch algorithm if the result is elementary.

The following example demonstrates the use of generating functions in solving linear recurrence relations.

Example 3.2 (Problem solving using GFs)**.** *Find the closed-form formula of u_n where*

$$\begin{cases} u_0 = v_1 = 1 \\ v_0 = u_1 = 0 \\ u_n = 2v_{n-1} + u_{n-2} \\ v_n = u_{n-1} + v_{n-2} \end{cases}$$

Let $U(z)$ and $V(z)$ be the generating function of the two sequences, then

$$\begin{cases} U(z) = 1 + 2zV(z) + z^2U(z) \\ V(z) = zU(z) + z^2U(z) = \frac{z}{1-z^2}U(z) \end{cases}$$

Solve the equation system to get the closed-form of $U(z)$, which is a rational function:

$$U(z) = \frac{1 - z^2}{1 - 4z^2 + z^4} = \frac{1}{3 - \sqrt{3}} \cdot \frac{1}{1 - \left(2 + \sqrt{3}\right) z^2} + \frac{1}{3 + \sqrt{3}} \cdot \frac{1}{1 - \left(2 - \sqrt{3}\right) z^2}$$

Thus

$$U_{2n+1} = 0 \qquad U_{2n} = \left\lceil \frac{(2 + \sqrt{3})^n}{3 - \sqrt{3}} \right\rceil$$

Generating functions of sequences can be extended to represent higher-dimensional sequences. For example, the generating function of the 2D sequence $a_{n,m}$ with meta-variables x and y is

$$G(a_{n,m}; x, y) = \sum_n \sum_m a_{n,m} x^n y^m$$

In general, the generating function for a k-dimensional sequence $f : \mathbb{N}^k \to \mathbb{R}$ has the following form

$$G(f; \mathbf{x}) = \sum_{\sigma \in \mathbb{N}^k} f(\sigma) \mathbf{x}^\sigma$$

3.2 Probability Generating Functions

The PGF of a non-negative discrete random variable X is defined as

$$g_X(t) = \mathbb{E}(t^X) = \sum_{n=0}^{\infty} \Pr(X = n) t^n$$

It can be regarded as the Z-transform of the probability mass function. PGFs provide a concise representation of probability distributions, greatly simplifying probabilistic reasoning. Table 2 summarizes PGFs of some frequently used distributions.

Distribution	Probability Mass Function	PGF
Bernoulli(p)	$\Pr(X = 1) = p,\ \Pr(X = 0) = 1 - p$	$1 - p + pz$
Binomial(n, p)	$\Pr(X = k) = \binom{n}{k} p^k (1 - p)^{n-k}$	$(1 - p + pz)^n$
Geometric(p)	$\Pr(X = k) = (1 - p)^{k-1} p$	$\frac{pz}{1-(1-p)z}$
Poisson(λ)	$\Pr(X = k) = \frac{\lambda^k e^{-\lambda}}{k!}$	$e^{\lambda(z-1)}$
DUnif(a, b)	$\Pr(X = k) = \frac{1}{b-a+1}$ for $k = a, \ldots, b$	$\frac{z^a(1-z^{b-a+1})}{(b-a+1)(1-z)}$
NBin(r, p)	$\Pr(X = k) = \binom{k+r-1}{k} (1 - p)^r p^k$	$\left(\frac{pz}{1-(1-p)z}\right)^r$

Table 2: PGFs of Common Distributions

Theorem 3.3 (PGF of the linear transform of a variable)**.** *Let X be a random variable whose PGF is $g_X(t)$. Then the PGF of $Y = aX + b$ is $g_Y(t) = t^b g_X(t^a)$*

$$g_Y(t) = \mathbb{E}(t^{aX+b}) = \mathbb{E}((t^a)^X \cdot t^b) = t^b \mathbb{E}((t^a)^X) = t^b g_X(t^a)$$

Theorem 3.4 (PGF of the sum of two independent variables)**.** *Let $g_X(t)$ and $g_Y(t)$ be the generating function of two independent random variables X and Y, respectively. Then the PGF of $Z = X + Y$ is $g_Z(t) = g_X(t) g_Y(t)$*

$$g_Z(t) = \mathbb{E}(t^Z) = \mathbb{E}(t^X t^Y) = \mathbb{E}(t^X)\mathbb{E}(t^Y) = g_X(t) g_Y(t)$$

Theorem 3.5 (PGF of random stopping sum)**.** *Let $X_1, X_2, \ldots$ be a sequence of iid variables with PGF $g_X(\cdot)$, and N be a random variable with PGF $g_N(\cdot)$. If N and $X_1, X_2, \ldots$ are independent, then the PGF* *of* $S = \sum_{i=1}^{N} X_i$ *is* $g_S = g_N \circ g_X$.

$$\begin{aligned}\mathbb{E}_{N,\mathbf{X}}\left\{t^{\sum_{i=1}^{N} X_i}\right\} &= \mathbb{E}_N\left\{\mathbb{E}_{\mathbf{X}|N}\left[t^{\sum_{i=1}^{N} X_i} \mid N\right]\right\} = \mathbb{E}_N\left\{\mathbb{E}_{\mathbf{X}|N}\left[\prod_{i=1}^{N} t^{X_i} \mid N\right]\right\} \\ &= \mathbb{E}_N\left\{\prod_{i=1}^{N} \mathbb{E}_{\mathbf{X}|N}\left[t^{X_i} \mid N\right]\right\} = \mathbb{E}_N\left\{\prod_{i=1}^{N} \mathbb{E}_{\mathbf{X}}\left[t^{X_i}\right]\right\} = \mathbb{E}_N\left\{(g_X(t))^N\right\} \\ &= \sum_{n=0}^{\infty} (G_X(t))^n \Pr(N = n) = g_N(g_X(t))\end{aligned}$$

Finally, PGFs can be generalized to work with random vectors. For example, the PGF of $(X, Y) \sim p(x, y)$ is

$$g_{X,Y}(s,t) = \mathbb{E}(s^X t^Y) = \sum_{n=0}^{\infty}\sum_{m=0}^{\infty} \Pr(X = n, Y = m) s^n t^m$$

In general, for a k-dimensional random vector $\mathbf{X} = (X_1, X_2, \ldots X_k)$, the PGF is defined as

$$g_{\mathbf{X}}(\mathbf{t}) = \mathbb{E}(\mathbf{t}^{\mathbf{X}}) = \sum_{\sigma \in \mathbb{N}^k} \Pr(\mathbf{X} = \sigma)\mathbf{t}^{\sigma}$$

4 PGF transformer semantics

Small-step operational semantics provide a micro-level view of program execution, focusing on the state evolution over one particular execution path. While this approach is suitable for deriving fine-grained program properties, it lacks the capability to capture the program's overall behavior. Denotational semantics, on the other hand, offers a holistic view by representing collections program states with mathematical objects and mapping program execution to transformations of these objects. The PGF transformer semantics represent the distribution over all reachable program states as a PGF. The semantics of a program is then modeled as a transformer that maps one PGF to another PGF.

Example 4.1 (PGF of distribution over program states)**.** *Consider the following ReDiP program*

$$N := 1; M :\approx \text{Poisson}(1/2)$$

For any initial program state distribution, the terminal program state distribution is always

$$p(n,m) = [m = 3]\binom{9}{n}\frac{1}{2^n \times 2^{9-n}}$$

In PGF transformer semantics, the program transforms any input state distribution PGF $g = \mathbb{E}(s^N t^M)$ *into* $\left(\frac{1}{2} + \frac{1}{2}s\right)^s t^3$*, denoted as*

$$[\![N := 1; M :\approx \text{Poisson}(1/2)]\!] = g \mapsto \left(\frac{1}{2} + \frac{1}{2}s\right)^s t^3$$

4.1 PGF transformer of loop-free ReDiP program

The PGF transformer semantics of ReDiP programs is defined as follows:

Definition 4.2 (PGF transformer of a ReDiP program)**.** *For a ReDiP program* P*, let* $(X_1, X_2, \ldots, X_k)$ *be all the variables used in a* P*, then the PGF transformer* $[\![P]\!]$ *is a* PGF → PGF *mapping such that:*

1. *For every distribution* μ_0 *over the possible program state* $\mathbb{N}^k$*. Let* $g(\mathbf{t}) = \mathbb{E}_{\mathbf{X}\sim\mu_0}(\mathbf{t}^{\mathbf{X}})$ *be the PGF of* $\mathbf{X} \sim \mu_0$.
2. *If the initial program state* σ_0 *is sampled from* μ_0*:* $\sigma_0 \sim \mu_0$.
3. *The terminal program state distribution* $\mu_1(\sigma_1) = \mathbb{E}_{\sigma_0\sim\mu_0}\left[TD(C, \sigma_0; \sigma_1)\right]$ *should satisfy* $[\![P]\!](g) = h$ *where* $h(\mathbf{t}) = \mathbb{E}_{\mathbf{Y}\sim\mu_1}(\mathbf{t}^{\mathbf{Y}})$ *is the PGF of* $\mathbf{Y} \sim \mu_1$.

We now explain how to compute PGF transformers of ReDiP programs. For the sake of simplicity, we assume that the program contains only two variables x and y and the PGF of program state distribution is $g = \mathbb{E}(s^X t^Y) = \sum_{x,y} p(x,y) s^x t^y$.

Theorem 4.3 (PGF transformers of non-conditional pGCL commands)**.** *The PGF transformers for skip commands, linear arithmetic commands, and random sampling commands:*

- $\llbracket \text{skip} \rrbracket = id$ *since a skip command does not change the program states.*
- $\llbracket x := 0 \rrbracket = g \mapsto g[s/1]$ *since* $g[s/1] = \mathbb{E}(1^X t^Y) = \mathbb{E}(t^Y) = \mathbb{E}(s^0 t^Y)$
- $\llbracket x := x + n \rrbracket = g \mapsto s^n g$ *since* $s^n g = s^n \mathbb{E}(s^X t^Y) = \mathbb{E}(s^n s^X t^Y) = \mathbb{E}(s^{X+n} t^Y)$
- $\llbracket x := x + y \rrbracket = g \mapsto g[t/st]$ *since* $g[t/st] = \mathbb{E}(s^X (st)^Y) = \mathbb{E}(s^X s^Y t^Y) = \mathbb{E}(s^{X+Y} t^Y)$
- $\llbracket x := x - 1 \rrbracket = g \mapsto g[s/0] + s^{-1}(g - g[s/0])$ *since*
$$\mathbb{E}(s^{\max(0,X-1)} t^Y) = \sum_y p(0,y) s^0 t^y + \sum_{x \geq 1} \sum_y p(x,y) s^{x-1} t^y = g[s/0] + s^{-1}(g - g[s/0])$$
- $\llbracket x :\approx \mu \rrbracket = g \mapsto g[s/1] h(s)$ *where* $h(s) = \mathbb{E}_{Z \sim \mu}(s^Z)$.
$$g[s/1] h(s) = \mathbb{E}(t^Y) \mathbb{E}_{Z \sim \mu}(s^Z) = \mathbb{E}_{Z \sim \mu}(s^Z t^Y) = \mathbb{E}_{X \sim \mu}(s^X t^Y)$$
- $\llbracket x := x + \text{iid}(\mu, y) \rrbracket = g \mapsto g[t/th(s)]$ *where* $h(s) = \mathbb{E}_{Z \sim \mu}(s^Z)$.
$$\begin{aligned} g[t/th(s)] &= \mathbb{E}(s^X (th(s))^Y) = \mathbb{E}(s^X (h(s))^Y t^Y) \\ &= \mathbb{E}_{Z_1, Z_2 \ldots \overset{\text{iid}}{\sim} \mu}(s^X s^{\sum_{i=1}^Y} t^Y) = \mathbb{E}_{Z_1, Z_2 \ldots \overset{\text{iid}}{\sim} \mu}(s^{X + \sum_{i=1}^Y} t^Y) \end{aligned}$$

The PGF transformers of several other ReDiP language constructions which does not involve conditional branching can also be computed:

- $\llbracket C_1; C_2 \rrbracket = \llbracket C_2 \rrbracket \circ \llbracket C_1 \rrbracket$
- $\llbracket \{C_1\}[p]\{C_2\} \rrbracket = g \mapsto p \llbracket C_1 \rrbracket(g) + (1-p) \llbracket C_2 \rrbracket(g)$
- $\llbracket \text{loop}(n)\{C\} \rrbracket = \llbracket C \rrbracket \circ \llbracket C \rrbracket \cdots \llbracket C \rrbracket$ *where* $\llbracket C \rrbracket$ *is repeated* n *times.*
- $\llbracket x := x - n \rrbracket = \llbracket \text{loop}(n)\{x := x - 1\} \rrbracket$

To work with conditional expressions, we define the filtered PGFs.

Definition 4.4 (Filtered PGFs)**.** *For* $g = \mathbb{E}(s^X t^Y)$ *the* k*-th Taylor term of* g *with respect to* s*,* $tt(g, s, n)$*, is define as*
$$tt(g, s, n) = \frac{s^k}{k!} \left(\frac{\partial^k g}{\partial s^k}[s/0] \right)$$
Filtered PGF is defined as $g_\varphi = \mathbb{E}([\varphi(x,y)] s^X t^Y)$ *where* $\varphi(x,y) \in \{0,1\}$ *is called the filtering condition. The exclusive prefix filtered PGF:* $g_{x<n}$ *for* $n \in \mathbb{N}$ *can be computed as*
$$g_{x<n} = \sum_{x<n} \sum_y p(x,y) s^x t^y = \sum_{k<n} tt(g, s, k)$$
Based on $g_{x<n}$ *several other filtered PGFs can be computed: the inclusive prefix* $g_{x \leq n} = g_{x<n+1}$*, the exclusive suffix* $g_{x>n} = g - g_{x \leq n}$*, and the inclusive suffix* $g_{x \geq n} = g_{x>n-1}$*. Furthermore* $g_{2|x}$ *and* $g_{2 \nmid x} = g - g_{2|x}$ *can also be efficiently computed:*
$$g + g[s/(-s)] = \mathbb{E}(s^X t^Y) + \mathbb{E}((-s)^X t^Y) = \mathbb{E}((1 + (-1)^X) s^X t^Y) = \mathbb{E}(2[2 \mid X] s^X t^Y) = 2 g_{2|x}$$
For compositional propositions $\varphi \wedge \psi$*,* $\varphi \vee \psi$*, and* $\neg\varphi$*, their corresponding filtered PGF can be computed recursively:*

- *Negation* $g_{\neg\varphi} = g - g_\varphi$
- *Conjunction* $g_{\varphi \wedge \psi} = h_\psi$ *where* $h = g_\varphi$
- *Disjunction* $g_{\varphi \vee \psi} = g_{\neg(\neg\varphi \wedge \neg\psi)}$

Transformers of branching and looping commands can be computed by leveraging filtered PGFs:

Theorem 4.5 (PGF transformer of if commands)**.** $\llbracket \text{if}(\varphi)\{C_1\} \, \text{else}\{C_2\} \rrbracket = g \mapsto \llbracket C_1 \rrbracket(g_\varphi) + \llbracket C_2 \rrbracket(g_{\neg\varphi})$

4.2 PGF transformer of loops

Unbounded loops are characterized as least fixed points in denotational semantics. To define least fixed points, we need a partial order $(\sqsubseteq, \mathrm{PGF})$.

Definition 4.6 (Partial order of PGFs)**.** *Let f and g be the PGFs of two k-dimensional discrete random vectors, where*

$$f = \sum_{\sigma} f(\sigma)\mathbf{t}^{\sigma} \qquad g = \sum_{\sigma} g(\sigma)\mathbf{t}^{\sigma}$$

We say $f \sqsubseteq g$ if and only if $\forall \sigma . f(\sigma) \leq g(\sigma)$.

This partial order is shown to be a ω-complete partial order[2]. Now that the PGF space is equipped with a partial order, we characterize loops with fixed points.

Theorem 4.7 (PGF transformer of while commands)**.** $[\![\mathrm{while}(\varphi)\{C\}]\!] = \mu\, Q.\, \mathrm{if}(\varphi)\{C; Q\}$
That is, the least fixed point of the function $Q \mapsto \mathrm{if}(\varphi)\{C; Q\}$.

Intuitively, a while loop is equivalent to an infinitely nested branching tree, which ultimately converges if the loop always terminates.

```
while(cond){
    body;
}
```

```
if(cond){ body;
if(cond){ body;
if(cond){ body;
  ....
}
}
}
```

```
least_fix_point((Q) => {
    if (cond) {body; Q}
})
```

The least fixed point characterization of while loops completes the last piece of the PGF transformer semantics of ReDiP. However, a general algorithm that computes the least fixed does not exist. Instead of finding the fixed point, we require that the fixed point be given as part of the input. The only work left to be done is to check whether the given program Q is indeed the fixed point of $Q \mapsto \mathrm{if}(\varphi)\{C; Q\}$.

Definition 4.8 (Probabilistic termination)**.** *A pGCL program P is said to be universally almost-surely terminating (UAST), if for every initial configuration $\langle P, \sigma, 0, \epsilon, 1\rangle$ it evolves to a halting configuration, i.e., a configuration of the form $\langle \downarrow, \sigma', n, \theta, q\rangle$, with probability 1.*

Theorem 4.9 (Fixed point induction)**.** *Let* $\mathrm{while}(\varphi)\{C\}$ *be a ReDiP loop, then*

$$[\![\mathrm{if}(\varphi)\{C; Q\}\,\mathrm{else}\{\mathrm{skip}\}]\!] \sqsubseteq [\![Q]\!] \implies [\![\mathrm{while}(\varphi)\{C\}]\!] \sqsubseteq [\![Q]\!]$$

Furthermore, if the loop is UAST, then

$$[\![\mathrm{if}(\varphi)\{C; Q\}\,\mathrm{else}\{\mathrm{skip}\}]\!] = [\![Q]\!] \iff [\![\mathrm{while}(\varphi)\{C\}]\!] = [\![Q]\!]$$

That is the fixed point for a loop is the loop invariant.

To check whether Q is the least fixed point of $\mathrm{while}(\varphi)\{C\}$:

1. Check if the loop is UAST with an external tool.
2. Check if $[\![\mathrm{if}(\varphi)\{C; Q\}\,\mathrm{else}\{\mathrm{skip}\}]\!] = [\![Q]\!]$. The two programs are both loop-free ReDiP. We will discuss how to determine the equivalence of loop-free ReDiP programs later.

Theorem 4.9 is a special case of Park's induction. Detailed proof can be found in the PRODIGY paper[2].

4.3 Properties of ReDiP PGF transformers

In this section, we prove two important properties of ReDiP PGF transformers, which enable efficient equivalence-checking algorithms. The general approach for proving the properties of ReDiP PGF transformers is structure induction over the program.

Theorem 4.10 (linearity)**.** *For every ReDiP program P, and two PGFs f, g.*

$$\forall \lambda \in [0,1]. \llbracket P \rrbracket(\lambda f + (1-\lambda) g) = \lambda \llbracket P \rrbracket(f) + (1-\lambda)\llbracket P \rrbracket(g)$$

Proof. Without loss of generality, we consider only two-dimensional PGFs. For two PGFs f, g and two non-negative real numbers a, b where $a + b = 1$.

1. Base case: PGF transformers of atomic commands are linear.

   (a) $x := 0$ transformer $g \mapsto g[s/1]$

   $$(af + bg)[s/1] = a(f[s/1]) + b(g[s/1])$$

   (b) $x := x + n$ transformer $g \mapsto s^n g$

   $$(af + bg) \mapsto (af + bg)s^n = a(s^n f) + b(s^n g)$$

   (c) $x := x + y$ transformer $g \mapsto g[t/st]$

   $$(af + bg) \mapsto (af + bg)[t/st] = a(f[t/st]) + b(g[t/st])$$

   (d) $x :\approx \mu$ transformer $g \mapsto g[s/1]h(s)$ where $h(\cdot)$ is the PGF of μ.
   The transformer is the composition of $g \mapsto g[s/1]$ and $g \mapsto gh(s)$ which are both linear, so the resulting mapping is linear.

   (e) $x := \mathrm{iid}(\mu, y)$ transformer $g \mapsto g[s/1][t/th(s)]$ where $h(\cdot)$ is the PGF of μ.
   The transformer $g \mapsto g[t/th(s)]$ is linear

   $$(af + bg) \mapsto (af + bg)[t/th(s)] = a(f[t/th(s)]) + b(g[t/th(s)])$$

   The transformer $\llbracket x := \mathrm{iid}(\mu, y) \rrbracket$ is the composition of $g \mapsto g[s/1]$ and $g \mapsto g[t/th(s)]$ which are both linear, so the resulting mapping is linear.

2. Induction case: Suppose that P, Q are ReDiP programs whose PGF transformers are linear.

   (a) $P; Q$ transformer $g \mapsto \llbracket Q \rrbracket(\llbracket P \rrbracket(g))$.

   $$(af + bg) \mapsto \llbracket Q \rrbracket(a\llbracket P \rrbracket(f) + b\llbracket Q \rrbracket(g)) = a\llbracket Q \rrbracket(\llbracket P \rrbracket(g)) + b\llbracket Q \rrbracket(\llbracket P \rrbracket(g))$$

   (b) $\{P\}[p]\{Q\}$ transformer $g \mapsto p\llbracket P \rrbracket(g) + (1-p)\llbracket Q \rrbracket(g)$.

   $$\begin{aligned}(af + bg) &\mapsto p\llbracket P \rrbracket(af + bg) + (1-p)\llbracket Q \rrbracket(af + bg) \\ &= ap\llbracket P \rrbracket(f) + bp\llbracket P \rrbracket(g) + a(1-p)\llbracket Q \rrbracket(f) + b(1-p)\llbracket Q \rrbracket(g) \\ &= a\left(p\llbracket P \rrbracket(f) + (1-p)\llbracket Q \rrbracket(f)\right) + b\left(p\llbracket P \rrbracket(g) + b(1-p)\llbracket Q \rrbracket(g)\right)\end{aligned}$$

   (c) $\mathrm{if}(\varphi)\{P\}\,\mathrm{else}\{Q\}$ transformer $g \mapsto \llbracket P \rrbracket(g_\varphi) + \llbracket Q \rrbracket(g_{\neg\varphi})$
   We can verify that the PGF filtering operators are linear.

   $$\begin{aligned}(af + bg) &\mapsto \llbracket P \rrbracket(af_\varphi + bg_\varphi) + \llbracket Q \rrbracket(af_\varphi + bg_\varphi) \\ &= a\llbracket P \rrbracket(f_\varphi) + b\llbracket P \rrbracket(g_\varphi) + a\llbracket Q \rrbracket(f_{\neg\varphi}) + b\llbracket Q \rrbracket(g_{\neg\varphi}) \\ &= a\left(\llbracket P \rrbracket(f_\varphi) + \llbracket Q \rrbracket(f_{\neg\varphi})\right) + b\left(\llbracket P \rrbracket(g_\varphi) + \llbracket Q \rrbracket(g_{\neg\varphi})\right)\end{aligned}$$

   (d) $\mathrm{while}(\varphi)\{P\}$ transformer $\mu\, Q.\, \mathrm{if}(\varphi)\{P; Q\}$
   Intuitively, a while loop is an infinitely nested branching tree. Considering PGF transformers of if-branchings are linear, the PGF transformer of a nested branching tree should also be linear. Rigorous proof of this need to properties of ω-complete partial order. The proof is omitted and can be found in[2].

□

Definition 4.11 (rational closed-forms). *$f \in \mathrm{PGF}$ is said to be in rational closed-form if*

$$f = \frac{g}{h} = \frac{\sum_{\sigma \le a^k} g(\sigma)\mathbf{t}^\sigma}{\sum_{\sigma \le b^k} h(\sigma)\mathbf{t}^\sigma} \quad a, b \in \mathbb{N}$$

That is, the f is the quotient of two finite polynomials.

Theorem 4.12 (rational closed-form preservation). *For a loop-free ReDiP program P, and a rational-closed form PGF g/h, there exists two finite polynomials g', h' such that $\llbracket P \rrbracket(g/h) = g'/h'$.*

Proof. Without loss of generality, we consider only two-dimensional PGFs. For two PGFs f, g and two non-negative real numbers a, b where $a + b = 1$.

1. Base case: PGF transformers of atomic commands preserve rational-closed form. This is because rational-closed forms are closed under compositor and linear combination. Note that substituting a variable with a constant/monomial/rational function is essentially composing the input PGF with a rational function. The detailed case analysis is omitted.
2. Induction case: PGF transformers of compositional commands preserve rational-closed form. This is because rational-closed form preserving mappings is closed under composition and linear combination. The detailed case analysis is omitted.

□

Rational closed-forms are amenable to computer algebra systems, e.g., `sympy` and `GiNaC`. The rational closed-form preservation property allows $\llbracket P \rrbracket(g/h)$ for a loop-free program P to be efficiently computed. Furthermore, rational functions have simple normal forms (sort the monomials in lexicographical order), so deciding rational functions equivalence is very easy.

Finally, we note that the closed-form preservation property is of more theoretical interest than practical significance. Our tool has support for sampling from a PGF specified by a elementary function, so our tool is actually working on elementary closed-form rather than rational closed-form. Though computations involving arbitrary elementary functions do not have simple and clear complexity bound, computer algebra systems are often able to compute the results in a reasonable time (for example, 10 seconds).

5 Deciding Equivalence of loop-free ReDiP programs

As we have seen in the previous section, fixed point induction allows transforming loopy ReDiP programs into semantically equivalent loop-free programs when correct loop invariants are supplied. Thus, ReDiP equivalence problem can be reduced to loop-free ReDiP equivalence problem.

Now consider the problem of checking whether two loop-free ReDiP programs P, Q are equivalent. Suppose that the program state of both programs is a random vector $(X_1, X_2, \ldots X_k)$.

1. By definition 2.4, P and Q are equivalent if (1) is true.

$$\forall \sigma_0 . \forall \sigma . TD(P, \sigma_0; \sigma) \equiv TD(Q, \sigma_0; \sigma) \tag{1}$$

2. In PGF transformer semantics, (1) is equivalent to (2).

$$\forall g \in \mathrm{PGF}(\mathbb{N}^k) . \llbracket P \rrbracket(g) = \llbracket Q \rrbracket(g) \tag{2}$$

3. $\llbracket P \rrbracket$ and $\llbracket Q \rrbracket$ are linear, so $\llbracket P \rrbracket \equiv \llbracket Q \rrbracket$ if and only if $\forall h \in \mathcal{B} . \llbracket P \rrbracket(h) = \llbracket Q \rrbracket(h)$, where $\mathcal{B}$ is a basis of $\mathrm{PGF}(\mathbb{N}^k)$. The set of all monomials $\{\mathbf{t}^\mathbf{x} \mid \mathbf{x} \in \mathbb{N}^k\}$ is a basis since every PGF, as a formal power series, can be expressed as a linear combination of monomials. Thus, (2) is equivalent to (3)

$$\forall \mathbf{x} \in \mathbb{N}^k . \llbracket P \rrbracket(\mathbf{t}^\mathbf{x}) = \llbracket Q \rrbracket(\mathbf{t}^\mathbf{x}) \tag{3}$$

Note that $\mathbf{t}^\mathbf{x} = \sum_{\mathbf{y} \in \mathbb{N}^k} \delta_\mathbf{x}(\mathbf{y}) \mathbf{t}^\mathbf{y}$ is the PGF of the point-mass distribution $\delta_\mathbf{x}(\mathbf{y}) = [\mathbf{x} = \mathbf{y}]$.

4. The set of monomials is infinite, so an algorithm cannot verify (3) exhaustively. To address this, embed all k-dim point-mass PGFs into a $2k$-dim PGF by introducing k extra meta-variables.

$$\Delta(\mathbf{t};\mathbf{u}) = \sum_{x_1,x_2,\dots x_k} (u_1t_1)^{x_1}(u_2t_2)^{x_2}\cdots(u_kt_k)^{x_k} = \prod_{i=1}^{k}\left(\sum_{j=0}^{\infty}(t_iu_i)^j\right) = \prod_{i=1}^{k}(1-t_iu_i)^{-1}$$

Δ is a so-called second-order PGF (SOP) since it can be regarded as a formal power series with meta-variable $\mathbf{u}$, where the coefficient of $\mathbf{u}^{\mathbf{x}}$ is the PGF of $\delta_{\mathbf{x}}(\cdot)$.

$$\begin{aligned}\forall \mathbf{x}\in\mathbb{N}^k.[\![P]\!](\mathbf{t}^{\mathbf{x}}) = [\![Q]\!](\mathbf{t}^{\mathbf{x}}) &\iff \sum_{\mathbf{x}\in\mathbb{N}^k}[\![P]\!](\mathbf{t}^{\mathbf{x}})\mathbf{u}^{\mathbf{x}} = \sum_{\mathbf{x}\in\mathbb{N}^k}[\![Q]\!](\mathbf{t}^{\mathbf{x}})\mathbf{u}^{\mathbf{x}}\\ &\iff [\![P]\!]\left(\sum_{\mathbf{x}\in\mathbb{N}^k}\mathbf{t}^{\mathbf{x}}\mathbf{u}^{\mathbf{x}}\right) = [\![Q]\!]\left(\sum_{\mathbf{x}\in\mathbb{N}^k}\mathbf{t}^{\mathbf{x}}\mathbf{u}^{\mathbf{x}}\right)\\ &\iff [\![P]\!](\Delta(\mathbf{t};\mathbf{u})) = [\![Q]\!](\Delta(\mathbf{t};\mathbf{u}))\end{aligned}$$

Therefore, (3) is equivalent to (4).

$$[\![P]\!](\Delta) = [\![Q]\!](\Delta) \quad \text{where} \quad \Delta = \prod_{i=1}^{k}(1-t_iu_i)^{-1} \tag{4}$$

The equivalence checking problem is ultimately reduced to verify if (4) holds. $[\![P]\!](\Delta), [\![Q]\!](\Delta)$ can be computed in finite time since P, Q are loop-free and Δ is rational. Therefore, the loop-free ReDiP equivalence problem is decidable.

6 Evaluation

Statement of data availability: Our implementation, benchmark suite, and benchmark results are made publicly available. The artifacts are hosted on a Github repository https://github.com/hehelego/prodigy-replica.

6.1 Experiment setup

We implemented the aforementioned algorithm in about 500 lines of Python code. All algebraic computations are carried out by `sympy`[7], a famous long-established computer algebra system written in Python.

To test the correctness and measure the efficiency of our tool, we measured the running time of our tool on 15 pairs of programs available in the PRODIGY repository. The PRODIGY repository also contains benchmarks for symbolic Bayesian inference, which is unrelated to equivalence checking. To get insight into the scalability of our tool, we also made a benchmark of 10 programs with finite loops of $1, 2, 3, \dots 10$ iterations.

```
nat x; x := geometric(1/2);
loop(n){ // number of iterations
 if(x > 15){ {x := x+1}[1/2]{x := x-1} }{}
}
```

The experiments were conducted on a laptop equipped with an 8-core x86-64 CPU (model AMD RYZEN 4800U) and 16GB of DDR4 memory.

6.2 Results

The test results are summarized in table 3 and table 4. The results show that our tool can correctly and efficiently verify the equivalence for various practical ReDiP programs. However, the scalability is unsatisfying figure 2 suggests that the verification time may grow exponentially as the program size grows.

Test case	Expected result	Result	Time
dep_bern.pgcl	equivalent	equivalent	6.81s
dueling_cowboys_parameter.pgcl	equivalent	equivalent	4.61s
geometric.pgcl	equivalent	equivalent	2.62s
geometric_observe.pgcl	equivalent	equivalent	2.72s
geometric_observe_parameter.pgcl	equivalent	equivalent	4.90s
geometric_parameter.pgcl	equivalent	equivalent	4.45s
geometric_shifted.pgcl	equivalent	equivalent	6.18s
ky_die.pgcl	equivalent	equivalent	18.5s
ky_die_2.pgcl	equivalent	equivalent	3.96s
n_geometric.pgcl	equivalent	equivalent	2.37s
negative_binomial_parameter.pgcl	equivalent	equivalent	4.07s
random_walk.pgcl	equivalent	equivalent	4.46s
running_paper_example.pgcl	equivalent	equivalent	2.36s
trivial_iid.pgcl	equivalent	equivalent	1.85s
ky_die_parameter.pgcl	not equivalent	not equivalent	26.0s

Table 3: Correctness test

iterations	1	2	3	4	5	6	7	8	9	10
time	2.75s	10.90s	19.36s	30.84s	43.52s	58.54s	72.68s	90.30s	105.35s	122.72s

Table 4: Scalability test

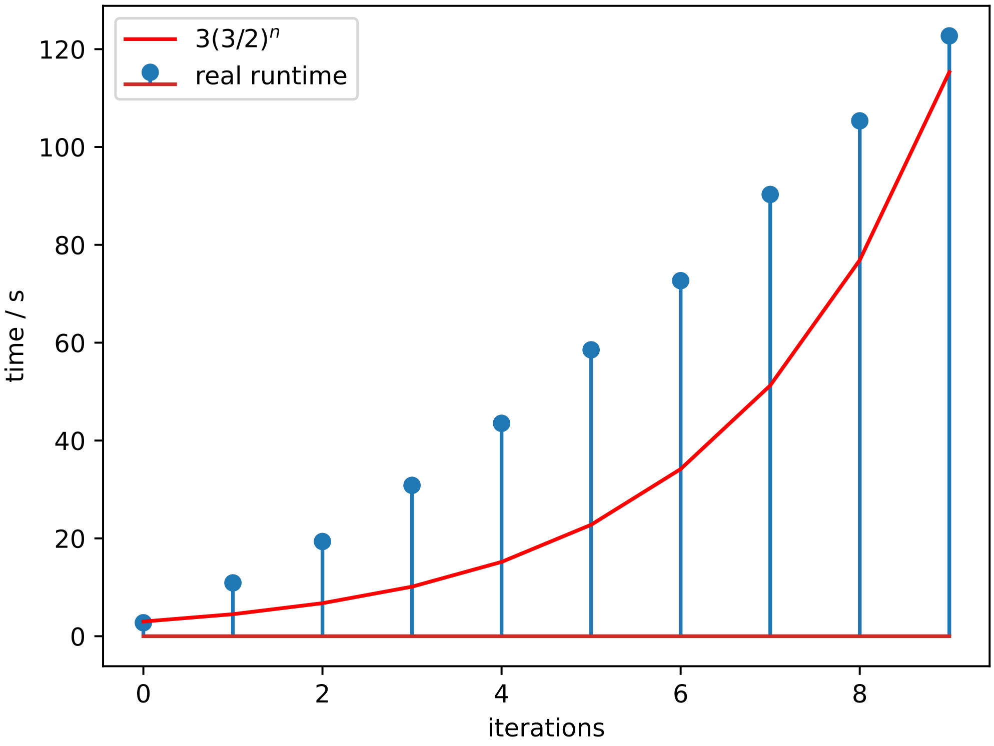

Figure 2: Visualization of scalability test result

6.3 Case Study: loop invariant checking

In this section, we give a step-by-step demonstration of the `n_geometric.pgcl` example.

First, consider the following ReDiP program. We can manually derive the PGF transformer of it. Let $g = \sum_{i=0}^{\infty} s^i h_i$, where $h_i = \frac{1}{i!}\frac{\partial^i g}{\partial s^i}[s/0]$, be the input PGF

```
$Input \leftarrow g = \mathbb{E}(s^n t^c)$
while$(n > 0)\{$$g_1 = g - g[s/0]$
  $\{n := n - 1\}$$g_2 = g_1 s^{-1}$
  $[1/2]$
  $\{c := c + 1\}$$g_3 = g_1 t$
  $g_4 = \frac{1}{2}(g_2 + g_3)$
$\}Output \leftarrow \mu\left[g \mapsto g[s/0] + \frac{1}{2}(s^{-1} + t)(g - g[s/0])\right]$
```

In this case, the least fixed point has a neat closed form:

1. Suppose f is the LFP, then $f = f[s/0] + \frac{1}{2}(s^{-1} + t)(f - f[s/0])$, which implies $f = f[s/0]$. Therefore, every term in the LFP is s-free.
2. The term $s^0 h_0$ never changes. As for $s^i h_i$, where $i > 0$:
   (a) It gets multiplied by $t/2$ or $s^{-1}/2$, until it becomes a s-free term.
   (b) Suppose that it becomes a s-free term after $i + j$ iterations: The last factor must be $s^{-1}/2$, and j factors among the other $i + j - 1$ factors are $t/2$. Therefore,
   $$s^i h_i \rightsquigarrow h_i 2^{-i}\binom{i+j-1}{j} 2^{-j} t^j = \binom{-i}{j} 2^{-j} t^j$$
   (c) Take the summation over $j = 0, 1, 2 \ldots$
   $$\frac{h_i}{2^i}\sum_{j=0}^{\infty}\binom{-i}{j} 2^{-j} t^j = \frac{h_i}{2^i}(1 - t/2)^{-i}$$
3. Thus, the least fixed point is
   $$g \mapsto h_0 + \sum_{i=1}^{\infty}\frac{h_i}{2^i}(1 - t/2)^{-i} = \sum_{i=0}^{\infty}\frac{h_i}{2^i}(1 - t/2)^{-i} = g[s/(1 - t/2)^{-1}]$$

Considering that $g \mapsto g[s/(1 - t/2)^{-1}] = [\![c := c + \text{iid}(\text{geometric}(1/2), n)]\!]$, the program should be semantically equivalent to the following loop-free ReDiP program denoted by I

```
$Input \leftarrow g = \mathbb{E}(s^n t^c)$
if$(n > 0)\{$$g_1 = g - g[s/0]$
  $c := c + \text{iid}(\text{geometric}(1/2), n);$ $g_2 = g_1[s/s(1 - t/2)^{-1}]$
  $n := 0$$g_3 = g_2[s/1] = g_1[s/(1 - t/2)^{-1}]$
$\}Output \leftarrow g[s/0] + (g - g[s/0])[s/(1 - t/2)^{-1}] = g[s/(1 - t/2)^{-1}]$
```

To check if the loopy version and the loop-free version are equivalent, we first apply fixed point induction to get the following program denoted by J

```
if($n > 0$){
  // loop body
  {$n := n - 1$}[1/2]{$c := c + 1$}
  // supplied loop invariant/fixed point
  if($n > 0$){
    $c := c + \mathrm{iid}(\mathrm{geometric}(1/2), n)$;
    $n := 0$
  }
}
```

We then check if $[\![I]\!](\Delta) = [\![J]\!](\Delta)$ where $\Delta = (1 - su)^{-1}(1 - tv)^{-1}$ for meta-variables $s - u$ and $t - v$. Our tool automatically gives the following results[1]

- Variables $\{c \mapsto u_0 x_0, n \mapsto u_1 x_1, tmp \mapsto u_2 x_2\}$.
- For the transformed program: `(-u1*(u2*x2 - 1) + (u2 - 1)*(u1 + x0 - 2)) / ((u2 - 1)*(u0*x0 - 1)*(u2*x2 - 1)*(u1 + x0 - 2))`
- For the spec program: `-u1/(u0*u1*u2*x0 - u0*u1*x0 + u0*u2*x0**2 - 2*u0*u2*x0 - u0*x0**2 + 2*u0*x0 - u1*u2 + u1 - u2*x0 + 2*u2 + x0 - 2) + 1/((u0*x0 - 1)*(u2*x2 - 1))`
- Outputs are equivalent.
- Finished in `2.320946780964732` seconds.

7 Conclusion and Discussion

So far, we have demonstrated the PGF transformer semantics of a simple probabilistic programming language, explained how the novel denotational semantics can be used to perform equivalence checking, built a tool for ReDiP equivalence verification, and tested it on small-scale inputs.

7.1 Limitations and Future Work

7.1.1 Conditional reasoning

Many probabilistic programming languages provide Bayesian inference features. For example, the following program should compute $\mathbb{E}(Z - X + Y \mid X^2 + Y^2 < Z^2)$ where $X, Y, Z \overset{\text{iid}}{\sim} \mathcal{N}(0, 1)$.

```
$x, y, z :\approx (normal(0,1), normal(0,1), normal(0,1))$;
observe($x^2 + y^2 < z^2$);
$s := z - (x + y)$;
$x := 0; y := 0; z := 0$;
```

How to augment the PGF transformer semantics to enable conditional reasoning is a future research direction of theoretical interest and practical value.

[1] The real `n_geometric.pgcl` uses an additional variable `tmp`

7.1.2 Incorporating Continuous Random Variables using MGF/CF

Many statistical models and stochastic processes involve real-valued random variables and practical probabilistic programming languages often support continuous random variables. However, ReDiP only supports integer-valued discrete random variables. It remains to explore how to add support for common distributions like $\mathcal{N}(\mu, \sigma^2)$ and $\beta(\alpha, \beta)$.

One possible attempt is to use characteristic functions (CF) or moment generating functions (MGF) instead of probability generating functions (PGF). CF/MGF enjoys nice properties like linearity and convolution theorem. Furthermore, CF of constant assignments and linear arithmetics can be easily expressed:

$$\begin{aligned}
\llbracket x := c \rrbracket &= g \mapsto c^s g[s/0] & e^{cs}\mathbb{E}(e^{0X+tY}) &= \mathbb{E}(e^{sc+tY}) \\
\llbracket x := x + y \rrbracket &= g \mapsto g[t/(s+t)] & \mathbb{E}(e^{sX+(t+s)Y}) &= \mathbb{E}(e^{s(X+Y)+tY}) \\
\llbracket x := ax \rrbracket &= g \mapsto g[s/as] & \mathbb{E}(e^{(sa)X+tY}) &= \mathbb{E}(e^{s(aX)+tY})
\end{aligned}$$

However, it remains unclear how to compute the prefix-filtered CF/MGF efficiently:

$$\varphi_{x<c}(t) = \mathbb{E}(t^{i[X<c]X}) = \int_{-\infty}^{c} f(x)e^{itx}\mathrm{d}x$$

7.1.3 Verifying Reactive Programs

Our approach relies on fixed point induction to handle unbounded loops. Fixed point induction requires the program to be UAST. Therefore, we can only correctly compute the PGF transformer of UAST programs.

Reactive programs (or streaming programs) process infinite data streams perpetually. These programs never terminate. For example, consider the following two programs. Our tool cannot determine whether they are equivalent or not.

```
setCallback((input) => {
  [a1, a2] := iid(std_normal, 2)
  x[n] := input;
  output(a1 * (x[n-1] - mu)
        + a2 * (x[n-2] - mu)^2);
  mu := (mu*n + x[n]) / (n+1);
  n = n + 1;
})
```

```
setCallback((input) => {
  [a1, a2] := iid(std_normal, 2)
  output(a1 * x[n-1]
        + a2 * x[n-2]^2);
  x[n] := input - mu;
  n := n + 1
  mu := (mu*n + x[n]) / n
})
```

In the future, one might research how PGF transformer semantics can be used to verify the equivalence of reactive programs.

References

[1] Chakarov, Aleksandar and Sankaranarayanan, Sriram. "Probabilistic program analysis with martingales". In: *Computer Aided Verification: 25th International Conference*. Springer. 2013, pp. 511–526.

[2] Mingshuai Chen et al. *Does a Program Yield the Right Distribution? Verifying Probabilistic Programs via Generating Functions.* 2022. arXiv: `2205.01449 [cs.LO]`.

[3] He Jifeng, K. Seidel, and A. McIver. "Probabilistic models for the guarded command language". In: *Science of Computer Programming* 28.2 (1997). Formal Specifications: Foundations, Methods, Tools and Applications, pp. 171–192. ISSN: 0167-6423. DOI: `https://doi.org/10.1016/S0167-6423(96)00019-6`. URL: `https://www.sciencedirect.com/science/article/pii/S0167642396000196`.

[4] Lutz Klinkenberg et al. "Exact Bayesian Inference for Loopy Probabilistic Programs using Generating Functions". In: *Proc. ACM Program. Lang.* 8.OOPSLA1 (2024).

[5] Lutz Klinkenberg et al. "Exact Probabilistic Inference Using Generating Functions". In: *LAFI.* [Extended Abstract]. 2023.

[6] Dexter Kozen. “Semantics of probabilistic programs”. In: *20th Annual Symposium on Foundations of Computer Science (sfcs 1979)*. IEEE. 1979, pp. 101–114.

[7] Aaron Meurer et al. “SymPy: symbolic computing in Python”. In: *PeerJ Computer Science* 3 (Jan. 2017), e103. ISSN: 2376-5992. DOI: `10.7717/peerj-cs.103`. URL: `https://doi.org/10.7717/peerj-cs.103`.

[8] Chen Mingshuai et al. “Does a program yield the right distribution? Verifying probabilistic programs via generating functions”. In: *International Conference on Computer Aided Verification*. Springer. 2022, pp. 79–101.

[9] Philipp Schröer, Lutz Klinkenberg, and Leo Mommers. *Probably*. `https://github.com/Philipp15b/probably`. 2021.

[10] Di Wang, Jan Hoffmann, and Thomas Reps. “Central moment analysis for cost accumulators in probabilistic programs”. In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 2021, pp. 559–573.

[11] Jinyi Wang et al. “Quantitative analysis of assertion violations in probabilistic programs”. In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2021. Virtual, Canada: Association for Computing Machinery, 2021, pp. 1171–1186. ISBN: 9781450383912. DOI: `10.1145/3453483.3454102`.

[12] Herbert S Wilf. *generatingfunctionology*. CRC press, 2005.