

# PGF transformer semantics

Cheng Peng   Dantong Liu<sup>1</sup>

June 4, 2024

---

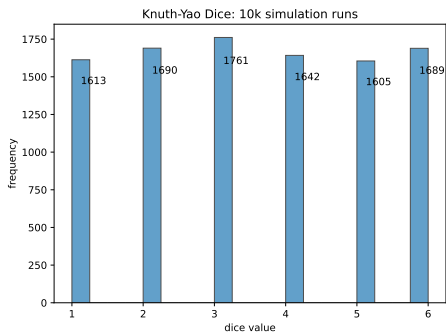
<sup>1</sup>External collaborator, BME undergrad at ShanghaiTech

# Motivating Example: Knuth-Yao Dice I

```
nat s; nat die;

while (s < 7) {
  if(s = 0) { { s:=1 }[1/2]{ s:=2 } }
  else { if(s = 1) { { s:=3 }[1/2]{ s:=4 } }
  else { if(s = 2) { { s:=5 }[1/2]{ s:=6 } }
  else { if(s = 3) { { s:=1 }[1/2]{ s:=7; die:=1 } }
  else { if(s = 4) { { s:=7; die:=2 }[1/2]{ s:=7; die:=3 } }
  else { if(s = 5) { { s:=7; die:=4 }[1/2]{ s:=7; die:=5 } }
  else { if(s = 6) { { s:=2 }[1/2]{ s:=7; die:=6 } }
  else { skip } } } } } }
```

# Motivating Example: Knuth-Yao Dice II



```
die := unif(1,6);
```

PGF transform semantics based equivalence checking of pGCL programs.

- ① A denotational semantics capturing the dynamics of all possible executions.
- ② The specification will be given as a pGCL program.
- ③ Focus on a linear fragment of pGCL, called ReDiP.

## Remarks

- Equivalence checking of pGCL is undecidable
- Linearity. No normalization is involved.

## 1 Introduction

- Backgrounds
- Limitations of previous works

## 2 Overview

## 3 pGCL and ReDiP

- Syntax and Semantics of pGCL
- Linear fragment of pGCL, ReDiP

## 4 PGF transformer semantics

- Review on PGFs
- PGF transformer semantics of ReDiP

- Fixed point induction
- Properties of the PGF transformer semantics
- PGF transformer equivalence

## 5 Evaluation

- Setup
- Results
- Case Study

## 6 Discussion

- Conclusion of this project
- Future works

## The rise of probabilistic programming

Randomness is ineluctable.

- Simulating the physics world.
- Building statistical models.
- Deriving efficient approximate algorithms.

## The need to verify stochastic programs

Your property and life are at stake.

- Security-critical applications: cryptography systems
- Safety-critical applications: cyber-physics systems

A trust problem: simulation can demonstrate unreliability but not prove reliability.

## matured approaches

- Evaluating moments of variables[9]
- Deriving assertion violation probability[10]
- Establishing lower/upper bounds[1]

## insufficiency

- Sensitivity to slight perturbation.
- Expressiveness: marginal/conditional. tail/concentration. parameter synthesis.

Our goal: to enable precise and versatile verification

## 1 Introduction

- Backgrounds
- Limitations of previous works

## 2 Overview

## 3 pGCL and ReDiP

- Syntax and Semantics of pGCL
- Linear fragment of pGCL, ReDiP

## 4 PGF transformer semantics

- Review on PGFs
- PGF transformer semantics of ReDiP

- Fixed point induction

- Properties of the PGF transformer semantics

- PGF transformer equivalence

## 5 Evaluation

- Setup
- Results
- Case Study

## 6 Discussion

- Conclusion of this project
- Future works



**Input program** A program in ReDiP, a fragment of pGCL

**Specification program** A loop-free ReDiP program: distribution of the final program state

**Loop invariants** Fixed point of loops, encoded as ReDiP programs.

**Verifier** check semantic equivalence, where the semantics of a program is

- A computational tree of configurations and transitions.
- A mapping from initial configuration to a distribution of output states.
- A (linear) transformer maps a PGF to another PGF.

**Output** True/False.

## 1 Introduction

- Backgrounds
- Limitations of previous works

## 2 Overview

## 3 pGCL and ReDiP

- Syntax and Semantics of pGCL
- Linear fragment of pGCL, ReDiP

## 4 PGF transformer semantics

- Review on PGFs
- PGF transformer semantics of ReDiP

- Fixed point induction
- Properties of the PGF transformer semantics
- PGF transformer equivalence

## 5 Evaluation

- Setup
- Results
- Case Study

## 6 Discussion

- Conclusion of this project
- Future works

$C \longrightarrow \text{skip}$	(effectless program)
$\quad   \text{diverge}$	(freeze)
$\quad   x := E$	(assignment)
$\quad   x \approx \mu$	(random assignment)
$\quad   C \circ C$	(sequential composition)
$\quad   \text{if}(\varphi)\{C\} \text{ else } \{C\}$	(conditional choice)
$\quad   \{C\} \square \{C\}$	(nondeterministic choice)
$\quad   \{C\} [p] \{C\}$	(probabilistic choice)
$\quad   \text{while}(\varphi)\{C\},$	(while loop)

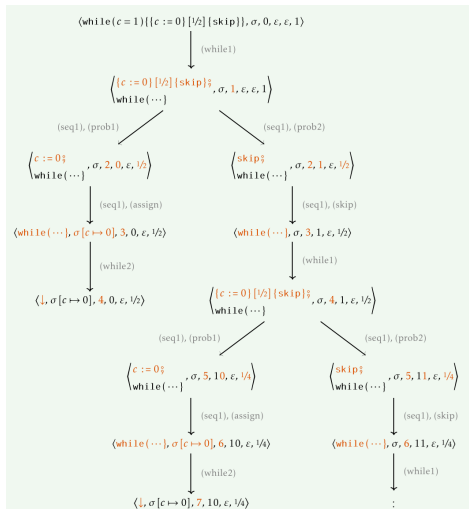
**configuration**  $\mathcal{K} = \langle C, \sigma, n, \theta, \eta, q \rangle$ .

**transition relation**  $\vdash \subseteq \mathbb{K} \times \mathbb{K}$ .

**computational tree** (initial configurations, reachable configuration, transitions).

# Syntax and Semantics of pGCL[3] II

$\frac{}{\langle \text{skip}, \sigma, n, \theta, \eta, q \rangle \vdash \langle \downarrow, \sigma, n+1, \theta, \eta, q \rangle}$ (skip)	
$\frac{}{\langle \text{diverge}, \sigma, n, \theta, \eta, q \rangle \vdash \langle \text{diverge}, \sigma, n+1, \theta, \eta, q \rangle}$ (diverge)	
$\frac{v = \sigma(E)}{\langle x := E, \sigma, n, \theta, \eta, q \rangle \vdash \langle \downarrow, \sigma[x \mapsto v], n+1, \theta, \eta, q \rangle}$ (assign)	
$\frac{\mu(\sigma)(v) = a > 0 \quad \aleph^{-1}(v) = i}{\langle x \approx \mu, \sigma, n, \theta, \eta, q \rangle \vdash \langle \downarrow, \sigma[x \mapsto v], n+1, \theta, i, \eta, q \cdot a \rangle}$ (rnd-assign)	
$\frac{\langle C_1, \sigma, n, \theta, \eta, q \rangle \vdash \langle C'_1, \sigma', n+1, \theta', \eta', q' \rangle \quad C'_1 \neq \downarrow}{\langle C_1 \S C_2, \sigma, n, \theta, \eta, q \rangle \vdash \langle C'_1 \S C_2, \sigma', n+1, \theta', \eta', q' \rangle}$ (seq1)	
$\frac{\langle C_1, \sigma, n, \theta, \eta, q \rangle \vdash \langle \downarrow, \sigma', n+1, \theta', \eta', q' \rangle}{\langle C_1 \S C_2, \sigma, n, \theta, \eta, q \rangle \vdash \langle C_2, \sigma', n+1, \theta', \eta', q' \rangle}$ (seq2)	
$\frac{\varphi(\sigma) = \text{true}}{\langle \text{if } (\varphi) \{ C_1 \} \text{ else } \{ C_2 \}, \sigma, n, \theta, \eta, q \rangle \vdash \langle C_1, \sigma, n+1, \theta, \eta, q \rangle}$ (if1)	
$\frac{\varphi(\sigma) = \text{false}}{\langle \text{if } (\varphi) \{ C_1 \} \text{ else } \{ C_2 \}, \sigma, n, \theta, \eta, q \rangle \vdash \langle C_2, \sigma, n+1, \theta, \eta, q \rangle}$ (if2)	
$\frac{}{\langle \{ C_1 \} \sqcap \{ C_2 \}, \sigma, n, \theta, \eta, q \rangle \vdash \langle C_1, \sigma, n+1, \theta, \eta L, q \rangle}$ (nondet1)	
$\frac{}{\langle \{ C_1 \} \sqcap \{ C_2 \}, \sigma, n, \theta, \eta, q \rangle \vdash \langle C_2, \sigma, n+1, \theta, \eta R, q \rangle}$ (nondet2)	
$\frac{p(\sigma) = a}{\langle \{ C_1 \} [p] \{ C_2 \}, \sigma, n, \theta, \eta, q \rangle \vdash \langle C_1, \sigma, n+1, \theta 0, \eta, q \cdot a \rangle}$ (prob1)	
$\frac{p(\sigma) = a}{\langle \{ C_1 \} [p] \{ C_2 \}, \sigma, n, \theta, \eta, q \rangle \vdash \langle C_2, \sigma, n+1, \theta 1, \eta, q \cdot (1-a) \rangle}$ (prob2)	
$\frac{\varphi(\sigma) = \text{true}}{\langle \text{while } (\varphi) \{ C \}, \sigma, n, \theta, \eta, q \rangle \vdash \langle C \S \text{while } (\varphi) \{ C \}, \sigma, n+1, \theta, \eta, q \rangle}$ (while1)	
$\frac{\varphi(\sigma) = \text{false}}{\langle \text{while } (\varphi) \{ C \}, \sigma, n, \theta, \eta, q \rangle \vdash \langle \downarrow, \sigma, n+1, \theta, \eta, q \rangle}$ (while2)	



- No divergence statement.
- All arithmetic expressions are linear/affine  
 $a_0 + a_1x_1 + a_2x_2 \cdots a_nx_n$  where  $a_i \in \mathbb{Z} \cup \text{Param}$  and  $x_i \in \text{Var}$ .
- All comparison expressions are rectangular  
 $\langle \text{expr} \rangle \text{ op } n$  where  $\text{op} \in \{=, \neq, >, <, \leq, \geq\}$  and  $n \in \mathbb{Z} \cup \text{Param}$ .
- Special support for  $x \bmod 2 = 0$ .
- Special support for IID sampling.
- Sampling from a user-defined PGF.

Still quite expressive :).

## 1 Introduction

- Backgrounds
- Limitations of previous works

## 2 Overview

## 3 pGCL and ReDiP

- Syntax and Semantics of pGCL
- Linear fragment of pGCL, ReDiP

## 4 PGF transformer semantics

- Review on PGFs
- PGF transformer semantics of ReDiP

- Fixed point induction

- Properties of the PGF transformer semantics

- PGF transformer equivalence

## 5 Evaluation

- Setup
- Results
- Case Study

## 6 Discussion

- Conclusion of this project
- Future works

## Notations

- vectors  $\mathbf{x} = (x_1, x_2, \dots, x_k)$  and  $\sigma : \{1, \dots, k\} \mapsto \mathbb{N}$
- monomials  $\mathbf{x}^\sigma = x_1^{\sigma(1)} x_2^{\sigma(2)} \dots x_k^{\sigma(k)} = \prod_i x_i^{\sigma(i)}$
- polynomial rings  $\mathbb{R}[[X]]$ ,  $\mathbb{R}[[X, Y]]$ , and  $\mathbb{R}[[\mathbf{X}]]$

## GF: sequences as formal power series

$$\sum_{n=0}^{\infty} a_n x^n \quad \sum_{n=0}^{\infty} \sum_{m=0}^{\infty} b_{n,m} x^n y^m \quad \sum_{\sigma \in \mathbb{N}^k} f(\sigma) \mathbf{x}^\sigma$$

## finite representation of infinite objects: closed-forms

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

## sequence manipulation as algebraic operations

sequence manipulation	gf algebra
$a_{n-m}$	$x^m f$
$a_n + b_n$	$f + g$
$\alpha a_n$	$\alpha f$
$\sum_k a_k b_{n-k}$	$fg$
$na_n$	$x\partial_x f$
$a_{n-1}/n$	$\int f dx$

## wikipedia/generating function/application/example 3

$$\begin{cases} U_n = 2V_{n-1} + U_{n-2} \\ V_n = U_{n-1} + V_{n-2} \\ U_0 = U_1 = V_0 = V_1 = 1 \end{cases} \implies \begin{cases} U(z) = 1 + 2zV(z) + z^2U(z) \\ V(z) = zU(z) + z^2U(z) = \frac{z}{1-z^2}U(z) \end{cases}$$

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

$$U_{2n+1} = 0 \quad U_{2n} = \left\lfloor \frac{(2+\sqrt{3})^n}{3-\sqrt{3}} \right\rfloor$$



# Probability Generating Functions I

Z-transform of PMF.  $\mathbb{E}(s^X) = \sum_n p(n)s^n$  and  $\mathbb{E}(s^X t^Y) = \sum_{n,m} p(n, m)s^n t^m$ .

## use of PGF

- represent distributions using generating functions
- operating random variable is manipulating PGFs
- working in finite closed-form

## Random stopping sum

A  $X_1, X_2, \dots$  a sequence of iid variables with PGF  $g_X(\cdot)$ . Another independent random variable  $N$  with PGF  $g_N(\cdot)$ .

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

## A denotational semantics

- Distribution of program states represented as PGFs.
- Program executions transforms PGFs.
- A program is a PGF transformer  $\llbracket P \rrbracket : \text{PGF} \rightarrow \text{PGF}$

Suppose that the program state is  $g = \mathbb{E}(s^X t^Y)$  where  $X, Y \in \mathbb{N}$ :

- $x := n$  assignment:  $g \mapsto g[s/1]s^n$

$$\mathbb{E}(s^X t^Y) \rightarrow s^n \mathbb{E}(1^X t^Y) = s^n \mathbb{E}(t^Y) = \mathbb{E}(s^n t^Y)$$

- $x := x+n$  cumulation:  $g \mapsto gs^n$

$$\mathbb{E}(s^X t^Y) \rightarrow s^n \mathbb{E}(s^X t^Y) = \mathbb{E}(s^{X+n} t^Y)$$

- $x := x-1$  self-decrement:  $g \mapsto (g - g[s/0])s^{-1} + g[s/0]$

$$\mathbb{E}(s^X t^Y) \rightarrow s^{-1} \mathbb{E}(s^X t^Y) = \mathbb{E}(s^{X-1} t^Y)$$

- $x := x+y$  addition:  $g \mapsto g[t/st]$

$$\mathbb{E}(s^X t^Y) \rightarrow \mathbb{E}(s^X (st)^Y) = \mathbb{E}(s^{X+Y} t^Y)$$

- $x := D$  samples from distribution:  $g \mapsto g[s/1] \cdot [D](s)$  where the PGF of  $D$  is  $[D](r)$

$$\mathbb{E}(s^X t^Y) \rightarrow \mathbb{E}(s^D) \mathbb{E}(1^X t^Y) = \mathbb{E}(s^D t^Y)$$

- $x := \text{iid}(D, y)$  iid sampling:  $g \mapsto g[s/1][t/t[D](s)]$

$$\mathbb{E}(s^X t^Y) \rightarrow \mathbb{E}(1^X t^Y ([D](s))^Y) = \sum_{0 \leq m} ([D](s))^m t^m$$

- $\text{if}(x < n) \{P\} \text{ else } \{Q\}$  conditional branching:  $g \mapsto \llbracket P \rrbracket(g_{x < n}) + \llbracket Q \rrbracket(g - g_{x < n})$  where

$$g_{x < n} = \sum_{x < n} \sum_y p(x, y) s^x t^y = \sum_{i < n} \frac{s^i}{i!} \left( \frac{\partial^i}{\partial s^i} g \right) [s/0]$$

- $P; Q$  sequential composition:  $g \mapsto \llbracket Q \rrbracket(\llbracket P \rrbracket(g))$

make it sweet. syntactic sugar

$\{P\} [r] \{Q\}$	$g \mapsto r\llbracket P \rrbracket(g) + (1-r)\llbracket Q \rrbracket(g)$
$\text{loop}(n) \{P\}$	$\llbracket P \rrbracket(\llbracket P \rrbracket \cdots \llbracket P \rrbracket(g))$
$x := x-n$	$\text{loop}(n) \{x := x-1\}$
$\text{if}(p \wedge q) \{P\} \text{ else } \{Q\}$	$\text{if}(p) \{ \text{if}(q) \{P\} \text{ else } \{Q\} \} \text{ else } \{Q\}$
$\text{if}(p \vee q) \{P\} \text{ else } \{Q\}$	$\text{if}(p) \{P\} \text{ else } \{ \text{if}(q) \{P\} \text{ else } \{Q\} \}$
$\text{if}(\neg p) \{P\} \text{ else } \{Q\}$	$\text{if}(p) \{Q\} \text{ else } \{P\}$
$\text{if}(x \leq n) P \text{ else } Q$	$\text{if}(x < n+1) P \text{ else } Q$
$\text{if}(x > n) P \text{ else } Q$	$\text{if}(x \leq n) Q \text{ else } P$
$\text{if}(x \geq n) P \text{ else } Q$	$\text{if}(x < n) Q \text{ else } P$
$\text{if}(x = n) P \text{ else } Q$	$\text{if}(x \leq n \wedge x \geq n) P \text{ else } Q$
$\text{if}(x \neq n) P \text{ else } Q$	$\text{if}(\neg(x = n)) P \text{ else } Q$
$x \bmod 2 = 0$	$g \mapsto \frac{1}{2}(g[s/-s] + g)$
$x \bmod 2 = 1$	$g \mapsto \frac{1}{2}(g - g[s/-s])$

# The missing piece in PGF transformer semantics, loops

- ① A while loop  $\text{while}(\phi)\{P\}$
- ② A infinitely nested braching tree

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

- ③ A least fixed point  $\mu X.\text{if}(\phi)\{P; X\}$

## rigorous and complicated theoretical derivation

Not quite correct.

- The PRODIGY verifier only checks invariant instead of automatically infer one.
- Defining a  $\omega$ -complete partial order ( $\leq$ , PGF)
- For Almost Surely Termination (AST) programs:  $\llbracket \text{if}(\phi)\{P; X\} \rrbracket \leq \llbracket X \rrbracket$
- For Universally AST (UAST) programs:  $\llbracket \text{if}(\phi)\{P; X\} \rrbracket = \llbracket X \rrbracket$

For a well-formed ReDiP program  $P$ ,  $\llbracket P \rrbracket$  is a linear transform

- $x := x+n$  transformer  $g \mapsto s^n g$

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

- $x := x+y$  transformer  $g \mapsto g[t/st]$

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

- $x := 0$  transformer  $g \mapsto g[s/1]$

$$(af + bg)[t/1] = af[t/1] + bg[t/1]$$

- Other simple ReDiP program statements.
- By induction  $P;Q$  transformer  $g \mapsto \llbracket Q \rrbracket(\llbracket P \rrbracket(g))$ .

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

- Other compositional statements  $\text{if}(\text{cond})\{P\}\text{else}\{Q\}$  and  $\text{while}(\text{cond})\{P\}$ .

Rational closed-forms are amenable to computer algebra systems (e.g., sympy and GiNaC).

$$\frac{f}{g} = \frac{\sum_{i=0}^n \sum_{j=0}^m a_{ij} x^i y^j}{\sum_{i=0}^n \sum_{j=0}^m b_{ij} x^i y^j}$$

Rational closed-forms are closed under ReDiP transforms.

- $x := x+n$  transformer  $g \mapsto s^n g$

$$\frac{f}{h} \mapsto \frac{s^n f}{h}$$

- $x := x+y$  transformer  $g \mapsto g[t/st]$

$$\frac{f}{h} \mapsto \frac{f[t/st]}{h[t/st]}$$

- $x := 0$  transformer  $g \mapsto g[s/1]$

$$\frac{f}{h} \mapsto \frac{f[s/1]}{h[s/1]}$$

- Other simple ReDiP program statements.
- By induction  $P;Q$  transformer  $g \mapsto \llbracket Q \rrbracket(\llbracket P \rrbracket(g))$ .

$$f/h \mapsto \llbracket Q \rrbracket(\llbracket P \rrbracket(f/h)) = \llbracket Q \rrbracket(f_1/h_1) = f_2/h_2$$

- Other compositional statements  $\text{if}(\text{cond})\{P\}\text{else}\{Q\}$  and  $\text{while}(\text{cond})\{P\}$ .

For two ReDiP programs  $P_1, P_2$  whose state is  $g(X_1, X_2, \dots, X_n) \in \text{PGF}(\mathbb{N}^n)$

- Equivalence checking: for all valid generating function of distributions over  $\mathbb{N}^n$ .  
 $\forall g \in \text{PGF}(\mathbb{N}^n) \quad \llbracket P_1 \rrbracket(g) = \llbracket P_2 \rrbracket(g)$
- By linearity: for all pointmass distributions over  $\mathbb{N}^n$   
 $\forall (y_1, y_2, \dots, y_n) \in \mathbb{N}^n \quad \llbracket P_1 \rrbracket(X_1^{y_1} X_2^{y_2} \dots X_n^{y_n}) = \llbracket P_2 \rrbracket(X_1^{y_1} X_2^{y_2} \dots X_n^{y_n})$
- Ebbed all n-dim pointmass PGFs into a 2n-dim PGF  $\delta(\mathbf{X}; \mathbf{U})$   
 $\llbracket P_1 \rrbracket(\delta) = \llbracket P_2 \rrbracket(\delta)$  where

$$\delta = \sum_{y_1, y_2, \dots, y_n} (U_1 X_1)^{y_1} (U_2 X_2)^{y_2} \dots (U_n X_n)^{y_n} = \prod_{i=1}^n \left( \sum_j (X_i U_i)^j \right) = \prod_{i=1}^n (1 - X_i U_i)^{-1}$$



- ① Restricting syntax of pGCL to obtain ReDiP.
- ② Recursively defining the PGF transformer semantics of ReDiP.
- ③ Handling unbounded loops with fixed-point induction.
- ④ Equivalence checking leveraging linearity

- 1 Introduction
  - Backgrounds
  - Limitations of previous works
- 2 Overview
- 3 pGCL and ReDiP
  - Syntax and Semantics of pGCL
  - Linear fragment of pGCL, ReDiP
- 4 PGF transformer semantics
  - Review on PGFs
  - PGF transformer semantics of ReDiP
  - Fixed point induction
  - Properties of the PGF transformer semantics
  - PGF transformer equivalence
- 5 Evaluation
  - Setup
  - Results
  - Case Study
- 6 Discussion
  - Conclusion of this project
  - Future works

## Implementation

- About 500 nlocs of Python implementation.
- pGCL analysis using the probably package[8].
- Algebraic computation with sympy[6].

## Benchmark

15 pairs of equivalent programs from PRODIGY[7].

Table: Results on the 15 test cases

test case	time
dep_bern.pgcl	6.81s
dueling_cowboys_parameter.pgcl	4.61s
geometric.pgcl	2.62s
geometric_observe.pgcl	2.72s
geometric_observe_parameter.pgcl	4.90s
geometric_parameter.pgcl	4.45s
geometric_shifted.pgcl	6.18s
ky_die.pgcl	18.5s
ky_die_2.pgcl	3.96s
ky_die_parameter.pgcl	26.0s
n_geometric.pgcl	2.37s
negative_binomial_parameter.pgcl	4.07s
random_walk.pgcl	4.46s
running_paper_example.pgcl	2.36s
trivial_iid.pgcl	1.85s

Table: scalability test: finite loop handling

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

- Check result: equivalent. 7.340888474005624 seconds.
- Variables  $\{c \mapsto u_0x_0, s \mapsto u_1x_1, tmp \mapsto u_2x_2\}$  parameters  $\llbracket$
- $\llbracket P_1 \rrbracket(\delta)$ :  $(-u_1*(u_2*x_2 - 1)*(u_1*x_0*\sqrt{1 - x_0**2} - u_1*x_0 + x_0**2 - x_0*(u_1*\sqrt{1 - x_0**2} - u_1 + x_0) + 2*\sqrt{1 - x_0**2} - 2)/2 + (u_2 - 1)*(u_1*\sqrt{1 - x_0**2} - u_1 + x_0))/((u_2 - 1)*(u_0*x_0 - 1)*(u_2*x_2 - 1)*(u_1*\sqrt{1 - x_0**2} - u_1 + x_0))$
- $\llbracket P_2 \rrbracket(\delta)$ :  $-u_1*(\sqrt{1 - x_0**2} - 1)/(u_0*u_1*u_2*x_0*\sqrt{1 - x_0**2} - u_0*u_1*u_2*x_0 - u_0*u_1*x_0*\sqrt{1 - x_0**2} + u_0*u_1*x_0 + u_0*u_2*x_0**2 - u_0*x_0**2 - u_1*u_2*\sqrt{1 - x_0**2} + u_1*u_2 + u_1*\sqrt{1 - x_0**2} - u_1 - u_2*x_0 + x_0) + 1/((u_0*x_0 - 1)*(u_2*x_2 - 1))$
- Comparison  $\llbracket P_1 \rrbracket(\delta) - \llbracket P_2 \rrbracket(\delta) = 0$

```
nat s; nat c; nat tmp;
```

```
while(s > 0){
  {s := s+1} [1/2] {s := s-1}
  c := c+1
  tmp := 0
}
```

```
nat s; nat c; nat tmp;
```

```
if(s > 0){
  tmp := iid( ((1-(1-(c*c))^(1/2)))/c), s
  c := c + tmp
  tmp := 0; s := 0
} else { skip }
```

## 1 Introduction

- Backgrounds
- Limitations of previous works

## 2 Overview

## 3 pGCL and ReDiP

- Syntax and Semantics of pGCL
- Linear fragment of pGCL, ReDiP

## 4 PGF transformer semantics

- Review on PGFs
- PGF transformer semantics of ReDiP

- Fixed point induction

- Properties of the PGF transformer semantics

- PGF transformer equivalence

## 5 Evaluation

- Setup
- Results
- Case Study

## 6 Discussion

- Conclusion of this project
- Future works

Our contributions:

- We define ReDiP a simple yet expressive subset of pGCL.
- We present the PGF transformers semantics of ReDiP.
- We explain the properties of PGF transformers.
- We designed an algorithm for ReDiP equivalence checking based on PGF transformer semantics.
- We tested the effectiveness and efficiency of such tool.



## conditional equivalence checking

From  $\llbracket P_1 \rrbracket = \llbracket P_2 \rrbracket$  to  $\llbracket P_1 | \phi \rrbracket = \llbracket P_2 | \phi \rrbracket$

```
-- prog: a program that handles input x such that P(x) is true
-- cond: the P(x) predicate
restrict :: (a -> Bool) -> (a -> b) -> (a -> Maybe b)
restrict cond prog input = if cond input
                           then Just (prog input)
                           else Nothing
-- check equivalence: (restrict p f) (restrict p g)
```

Obstacles:

- Support for conditional reasoning (observe(..) in pGCL).
- Expressing conditions besides rectangular ones, e.g.,  $x^2 + y^2 < 1$ .

## incorporating real-valued variables using MGF/CF

Discrete random variables like Poisson( $\lambda$ ) and Binomial( $n, p$ ) are not enough. How to add support for  $\mathcal{N}(\mu, \sigma^2)$  and  $\beta(\alpha, \beta)$ ?

A natural generalization would be:

$$\mathbb{E}(e^{sX+tY}) \rightarrow e^{cs} \mathbb{E}(e^{0X+tY}) = \mathbb{E}(e^{sc+tY})$$

$$\mathbb{E}(e^{sX+tY})^* \rightarrow \mathbb{E}(e^{sX+(t+s)Y}) = \mathbb{E}(e^{s(X+Y)tY})$$

Obstacles:

- Rational closed-form preservation no longer holds.
- Rectangular bound  $x < c$  hard to express for MGF/CF

$$\mathbb{E}(t^{X|X < n}) = \sum_{x < n} p(x = n) t^n$$

$$\mathbb{E}(e^{tX|X < n}) = \int_{-\infty}^c f(x) e^{tx} dx$$

## reactive programs

- The current approach works for UAST programs.
- Reactive programs are naturally non-terminating.
- Possible solution: bisimulation based equivalence checking.

```
OnInputSymbol(()) => {  
  alpha1, alpha2 = iid(std_normal, 2)  
  x[n] = input();  
  output(alpha1 * (x[n-1] - mu)  
    + alpha2 * (x[n-2] - mu)^2);  
  mu = (mu * n + x[n]) / (n + 1);  
  n += 1;  
})
```

```
OnInputSymbol(()) => {  
  alpha1, alpha2 = iid(std_normal, 2)  
  output(alpha1 * (x[n-1] - mu)  
    + alpha2 * (x[n-2]^2  
    - 2*x[n]*mu + mu^2));  
  x[n] = input();  
  n += 1  
  mu = (mu * (n-1) + x[n]) / n  
})
```

- [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 2022. arXiv: 2205.01449 [cs.LG].
- [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](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] 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>.

- [7] 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.
- [8] Philipp Schröder, Lutz Klinkenberg, and Leo Mommers. Probably. <https://github.com/Philipp15b/probably>. 2021.
- [9] 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 2021, pp. 559–573.
- [10] Jinyi Wang et al. “Quantitative analysis of assertion violations in probabilistic programs”. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language PLDI 2021. Virtual, Canada: Association for Computing Machinery, 2021, pp. 1171–1186. ISBN: 9781450383912. DOI: 10.1145/3453483.3454102.
- [11] Herbert S Wilf. generatingfunctionology. CRC press, 2005.

Questions are welcomed.