Analysis of Probabilistic Programs

Pushing the Limits of Automation

Joost-Pieter Katoen

Software Modeling and Verification Group
RWTH Aachen University

with Hongfei Fu, Friedrich Gretz, Benjamin Kaminski
and Annabelle McIver, Larissa Meinecke, Carroll Morgan

September 17, 2014

Overview

Probabilistic programs

What are probabilistic programs?

Sequential, possibly non-deterministic, programs with random assignments.

Applications

Security, machine learning, quantum computing, randomized algorithms

The scientific challenge

- Such programs are small, but hard to understand and analyse [1].
- Problems: infinite variable domains, (lots of) parameters, and loops.

$\Rightarrow$ Our aim: **push the limits of automated analysis**

[1] Their analysis is undecidable.

Once upon a time . . . . . .

Duelling cowboys

```
int cowboyDuel(float a, b) { // 0 < a < 1, 0 < b < 1
   int t := A [] t := B; // decide who shoots first
   bool c := true;
   while (c) {
     if (t = A) {
         (c := false [a] t := B); // A shoots B with prob. a
     } else {
         (c := false [b] t := A); // B shoots A with prob. b
     }
   }
return t; // the survivor
}
```

Claim:

Cowboy A wins the duel with probability at least $\frac{(1-b)\cdot a}{a+b-a\cdot b}$.

Playing with geometric distributions

- X is a random variable, geometrically distributed with parameter p
- Y is a random variable, geometrically distributed with parameter q

Q: generate a sample x, say, according to the random variable $X - Y$

```
int XminY1(float p, q){ // 0 <= p, q <= 1
  int x := 0;
  bool flip := false;
  while (not flip) { // take a sample of X to increase x
    (x +:= 1 [p] flip := true);
  }
  flip := false;
  while (not flip) { // take a sample of Y to decrease x
    (x -:= 1 [q] flip := true);
  }
  return x; // a sample of X-Y
}
```

An alternative program

```
int XminY2(float p, q){
  int x := 0;
  bool flip := false;
  (flip := false [0.5] flip := true); // flip a fair coin
  if (not flip) {
    while (not flip) { // sample X to increase x
      (x +:= 1 [p] flip := true);
    }
  } else {
    flip := false; // reset flip
    while (not flip) { // sample Y to decrease x
      x -:= 1;
      (skip [q] flip := true);
    }
  }
return x; // a sample of X-Y
}
```

Program equivalence

```
int XminY1(float p, q){
  int  x, f := 0, 0;
  while (f = 0) {
    (x +:= 1 [p] f := 1);
  }
  f := 0;
  while (f = 0) {
    (x -:= 1 [q] f := 1);
  }
  return x;
}
```

```
int XminY2(float p, q){
  int x, f := 0, 0;
  (f := 0 [0.5] f := 1);
  if (f = 0) {
    while (f = 0) {
      (x +:= 1 [p] f := 1);
    }
  } else {
    f := 0;
    while (f = 0) {
      x -:= 1;
      (skip [q] f := 1);
    }
  }
return x;
}
```

Claim: [Kiefer *et al.*, 2012]

Both programs are equivalent for $(p, q) = (\frac{1}{2}, \frac{2}{3})$. Q: No other ones?

Analysing fully probabilistic programs is hard

How hard? **[Kaminski & Katoen, 2014]**

- Expected outcome exceeds a rational threshold — in Σ_1^0
- Expected outcome is below a rational threshold — Σ_2^0-complete
- Expected outcome equals a rational value — Π_2^0-complete
- Almost-sure termination — Π_2^0-complete

$\Rightarrow$ Almost-sure termination is harder than termination of ordinary programs

$\Rightarrow$ Almost-sure termination is not harder than the universal halting problem

Correctness of probabilistic programs

Question:

How to verify the correctness of such programs? In an automated way?

Apply model checking?

- Apply MDP model checking. LiQuor, PRISM, STORM
  - ⇒ works for program instances, but no general solution.
- Use abstraction-refinement techniques. PASS, POGAR
  - ⇒ loop analysis with real variables does not work well.
- Check language equivalence. APEX
  - ⇒ cannot deal with parameterised probabilistic programs.
- Apply parameterised probabilistic model checking. PARAM
  - ⇒ deals with fixed-sized probabilistic programs.

Apply deductive verification!

[McIver & Morgan]

Duelling cowboys

```
int cowboyDuel(float a, b) { // 0 < a < 1, 0 < b < 1
   int t := A [] t := B; // decide who shoots first
   bool c := true;
   while (c) {
     if (t = A) {
         (c := false [a] t := B); // A shoots B with prob. a
     } else {
         (c := false [b] t := A); // B shoots A with prob. b
     }
   }
return t; // the survivor
}
```

We can infer:

Cowboy A wins the duel with probability at least $\dfrac{(1-b)\cdot a}{a+b-a\cdot b}$.

Program equivalence

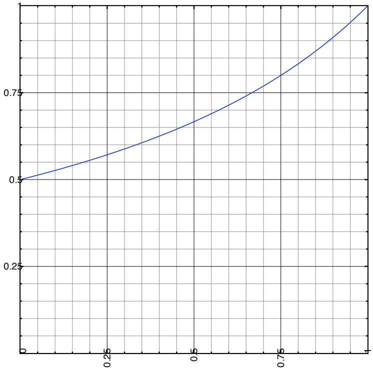
```
int XminY1(float p, q){
  int  x, f := 0, 0;
  while (f = 0) {
    (x +:= 1 [p] f := 1);
  }
  f := 0;
  while (f = 0) {
    (x -:= 1 [q] f := 1);
  }
  return x;
}
```

```
int XminY2(float p, q){
  int x, f := 0, 0;
  (f := 0 [0.5] f := 1);
  if (f = 0) {
    while (f = 0) {
      (x +:= 1 [p] f := 1);
    }
  } else {
    f := 0;
    while (f = 0) {
      x -:= 1;
      (skip [q] f := 1);
    }
  }
return x;
}
```

Our analysis yields:

Both programs are equivalent for any q with $q = \frac{1}{2-p}$.

Graphically this means . . .



Both programs yield the same expected outcome for all points on the curve $q = \frac{1}{2-p}$.

Roadmap of the talk

Overview

Dijkstra's guarded command language

`skip`	empty statement
`abort`	abortion
`x := E`	assignment
`prog1 ; prog2`	sequential composition
`if (G) prog1 else prog2`	choice
`prog1 [] prog2`	non-deterministic choice
`while (G) prog`	iteration

Probabilistic guarded command language pGCL


- **skip** — empty statement
- **abort** — abortion
- x := E — assignment
- prog1 ; prog2 — sequential composition
- **if** (G) prog1 **else** prog2 — choice
- prog1 [] prog2 — non-deterministic choice
- prog1 [p] prog2 — probabilistic choice
- **while** (G) prog — iteration

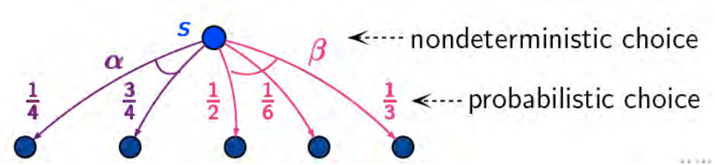
Overview

Markov decision processes

Markov decision process

An MDP $\mathcal{M}$ is a tuple $(S, S_0, \longrightarrow)$ where

- S is a countable set of states with initial state-set $S_0 \subseteq S$, $S_0 \neq \varnothing$
- $\longrightarrow \subseteq S \times Dist(S)$ is a transition relation



Operational semantics of pGCL

Aim: Model the behaviour of a program $P \in \texttt{pGCL}$ by an MDP $\mathcal{M}[\![P]\!]$.

Approach:

- Let η be a variable valuation of the program variables
- Use the special (semantic) construct exit for successful termination
- States are of the form $\langle Q, \eta \rangle$ with $Q \in \texttt{pGCL}$ or $Q = \text{exit}$
- Initial states are tuples $\langle P, \eta \rangle$ where η fulfils the initial conditions
- Transition relation $\rightarrow$ is smallest relation satisfying the SOS rules

Structured operational semantics

$$\langle \text{skip}, \eta \rangle \to \langle \text{exit}, \eta \rangle \qquad \langle \text{abort}, \eta \rangle \to \langle \text{abort}, \eta \rangle$$

$$\langle x := \text{expr}, \eta \rangle \to \langle \text{exit}, \eta[x := [\![\, \text{expr} \,]\!]_\eta] \rangle$$

$$\langle P [] Q, \eta \rangle \to \langle P, \eta \rangle \qquad \langle P [] Q, \eta \rangle \to \langle Q, \eta \rangle$$

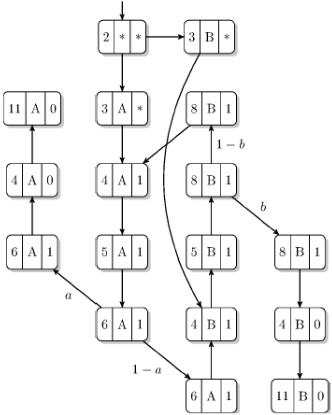
$$\langle P[p]Q, \eta \rangle \to \mu \text{ with } \mu(\langle P, \eta \rangle) = p \text{ and } \mu(\langle Q, \eta \rangle) = 1-p$$

$$\frac{\langle P, \eta \rangle \to \mu}{\langle P; Q, \eta \rangle \to \nu} \text{ with } \nu(\langle P'; Q', \eta' \rangle) = \mu(\langle P', \eta' \rangle) \text{ where exit}; P = P$$

$$\frac{\eta \models G}{\langle \text{while}(G)\{P\}, \eta \rangle \to \langle P; \text{while } (G)\{P\}, \eta \rangle} \qquad \frac{\eta \not\models G}{\langle \text{while}(G)\{P\}, \eta \rangle \to \langle \text{exit}, \eta \rangle}$$

MDP of duelling cowboys

```
int cowboyDuel(float a, b) {
  int t := A [] t := B;
  bool c := true;
  while (c) {
    if (t = A) {
      (c := false [a] t := B);
    } else {
      (c := false [b] t := A);
    }
  }
  return t;
}
```



This MDP is parameterized but finite. Once we count the number of shots before one of the cowboys dies, the MDP becomes infinite. Our approach however allows to determine e.g., the expected number of shots before success.

Overview

Weakest preconditions

Weakest precondition **[Dijkstra 1975]**

A predicate transformer is a total function between two predicates on the state of a program.

The predicate transformer $wp(P, F)$ for program P and postcondition F yields the "weakest" precondition E on the initial state of P ensuring that the execution of P terminates in a final state satisfying F.

Hoare triple $\{E\}\,P\,\{F\}$ holds for total correctness iff $E \Rightarrow wp(P, F)$.

Predicate transformer semantics of Dijkstra's GCL

Syntax	Semantics $wp(P, F)$
`skip`	F
`abort`	$false$
`x := E`	$F[x := E]$
`P1 ; P2`	$wp(P_1, wp(P_2, F))$
`if (G) P1 else P2`	$(G \Rightarrow wp(P_1, F)) \wedge (\neg G \Rightarrow wp(P_2, F))$
`P1 [] P2`	$wp(P_1, F) \wedge wp(P_2, F)$
`while (G)P`	$\mu X. ((G \Rightarrow wp(P, X)) \wedge (\neg G \Rightarrow F))$

μ is the least fixed point operator wrt. the ordering $\Rightarrow$ on predicates.

Expectations

Weakest pre-expectation [McIver & Morgan 2004]

An expectation maps program states onto non-negative reals. It's the quantitative analogue of a predicate.

An expectation transformer is a total function between two expectations on the state of a program.

The transformer $wp(P, f)$ for program P and post-expectation f yields the least expectation e on P's initial state ensuring that P's execution terminates with an expectation f.

Annotation $\{e\}\, P\, \{f\}$ holds for total correctness iff $e \leqslant wp(P, f)$, where $\leqslant$ is to be interpreted in a point-wise manner.

Expectation transformer semantics of pGCL

Syntax	Semantics $wp(P, f)$
`skip`	f
`abort`	0
`x := E`	$f[x := E]$
`P1 ; P2`	$wp(P_1, wp(P_2, f))$
`if (G) P1 else P2`	$[G] \cdot wp(P_1, f) + [\neg G] \cdot wp(P_2, f)$
`P1 [] P2`	$\min\left(wp(P_1, f), wp(P_2, f)\right)$
`P1 [p] P2`	$p \cdot wp(P_1, f) + (1-p) \cdot wp(P_2, f)$
`while (G)P`	$\mu X. ([G] \cdot wp(P, X) + [\neg G] \cdot f)$

μ is the least fixed point operator wrt. the ordering $\leqslant$ on expectations.

A simple slot machine

```
void flip {
  d1 := ♡ [1/2] ♢;
  d2 := ♡ [1/2] ♢;
  d3 := ♡ [1/2] ♢;
}
```

Example weakest pre-expectations

Let $all(x) \;\equiv\; (x = d_1 = d_2 = d_3)$.

- If $f = [all(\heartsuit)]$, then $wp(flip, f) = \frac{1}{8}$.
- If $g = 10 \cdot [all(\heartsuit)] + 5 \cdot [all(\diamondsuit)]$, then:

$$wp(flip, g) \;=\; \frac{15}{8} \;=\; 6 \cdot \frac{1}{8} \cdot 0 + 1 \cdot \frac{1}{8} \cdot 10 + 1 \cdot \frac{1}{8} \cdot 5$$

So the least fraction of the jackpot the gamer can expect to win is $\frac{15}{8}$.

Overview

MDPs with rewards

To compare the operational and wp- and wlp-semantics, we use rewards.

MDP with rewards

An MDP with rewards is a pair $(\mathcal{M}, r)$ with $\mathcal{M}$ an MDP with state space S and $r : S \to \mathbb{R}$ a function assigning a real reward to each state.

The reward $r(s)$ stands for the reward earned on entering state s.

Cumulative reward for reachability

Let $\pi = s_0 \xrightarrow{\mu_0} s_1 \xrightarrow{\mu_1} \ldots$ be an infinite path in $(\mathcal{M}, r)$ and $T \subseteq S$ a set of target states such that $\pi \models \Diamond T$. The cumulative reward along π before reaching T is defined by:

$$r_T(\pi) = r(s_0) + \ldots + r(s_k) \text{ where } s_i \notin T \text{ for all } i < k \text{ and } s_k \in T.$$

If $\pi \not\models \Diamond T$, then $r_T(\pi) = 0$.

Expected reward reachability

Expected reward for reachability

The minimal expected reward until reaching $T \subseteq S$ from $s \in S$ is:

$$MinER(s \models \Diamond T) = \min_{\mathfrak{P}} \int_0^\infty c \cdot Pr^{\mathfrak{P}}\{\, \pi \in Paths^{\mathfrak{P}}(s, \Diamond T) \mid r_T(\pi) = c \,\}\, dc$$

Positional policies suffice for expected reward objectives.

Relating operational and wp-semantics of pGCL

Weakest pre-expectations vs. expected reachability rewards

For pGCL-program P, variable valuation η, and post-expectation f:

$$wp(P, f)(\eta) \;=\; MinER^{\mathcal{M}[\![P]\!]}(\langle P, \eta \rangle \models \Diamond P^{\surd})$$

where rewards in MDP $\mathcal{M}[\![P]\!]$ are: $r(\langle \text{exit}, \eta' \rangle) = f(\eta')$ and 0 otherwise.

Thus, $wp(P, f)$ evaluated at η is the minimal expected value of f over any of the resulting distributions of P. The weakest liberal pre-expectation $wp(P, f)$ is similar taking into account the possibility of non-termination.

Overview

Qualitative loop invariants

Recall that for while-loops we have:

$$wp(\text{while}(G)\{P\}, F) \;=\; \mu X.\,(G \Rightarrow wp(P, X) \wedge \neg G \Rightarrow F)$$

To determine this *wp*, one exploits an "invariant" I such that $\neg G \wedge I \Rightarrow F$.

Loop invariant

Predicate I is a loop invariant if it is preserved by loop iterations:

$$G \wedge I \Rightarrow wp(P, I) \qquad \text{(consecution condition)}$$

Then: $\{I\}$ while$(G)\{P\}$ $\{F\}$ is a correct program annotation.

Linear invariant generation [Colón et al., 2003]

Linear programs

A program is linear if all guards are linear constraints, and updates are linear expressions (in the real program variables).

Approach by Colón et al.

1. Speculatively annotate a program with linear boolean expressions:

$$\alpha_1 \cdot x_1 + \ldots + \alpha_n \cdot x_n + \alpha_{n+1} \leqslant 0$$

   where α_i is a parameter and x_i a program variable.
2. Express verification conditions as inequality constraints over α_i, x_i.
3. Transform these inequality constraints into polynomial constraints (e.g., using Farkas lemma).
4. Use off-the-shelf constraint-solvers to solve them (e.g., REDLOG).
5. Exploit resulting assertions to infer program correctness.

Quantitative loop invariants

Recall that for while-loops we have:

$$wp(\text{while}(G)\{P\}, f) \;=\; \mu X.\ ([G] \cdot wp(P, X) + [\neg G] \cdot f)$$

To determine this *wp*, we use an "invariant" I such that $[\neg G] \cdot I \;\leqslant\; f$.

Quantitative loop invariant

Expectation I is a quantitative loop invariant if —by consecution—

- it is preserved by loop iterations: $[G] \cdot I \;\leqslant\; wlp(P, I)$.

To guarantee soundness, I has to fulfill either:

1. I is bounded from below and by above by some constants, or
2. on each iteration there is a probability $\epsilon > 0$ to exit the loop

Then: $\{I\}$ while$(G)\{P\}$ $\{f\}$ is a correct program annotation.

Our approach

Main steps

1. Speculatively annotate a program with linear expressions:

   $$[\alpha_1 \cdot x_1 + \ldots + \alpha_n \cdot x_n + \alpha_{n+1} \ll 0] \cdot (\beta_1 \cdot x_1 + \ldots + \beta_n \cdot x_n + \beta_{n+1})$$

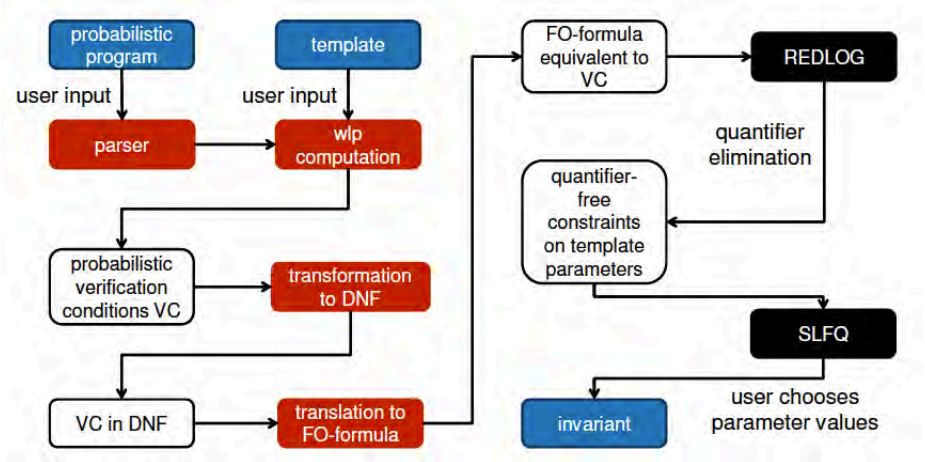
   with real parameters α_i, β_i, program variable x_i, and $\ll \in \{ <, \leqslant \}$.
2. Transform these numerical constraints into Boolean predicates.
3. Transform these predicates into non-linear FO formulas.
4. Use constraint-solvers for quantifier elimination (e.g., REDLOG).
5. Simplify the resulting formulas (e.g., using SLFQ and SMT solving).
6. Exploit resulting assertions to infer program correctness.

Soundness and completeness

Theorem

For any linear pGCL program annotated with propositionally linear expressions, our method will find all parameter solutions that make the annotation valid, and no others.

PRINSYS Tool: Synthesis of Probabilistic Invariants



download from `moves.rwth-aachen.de/prinsys`

Duelling cowboys: when does A win?

```
int cbDuel(float a, b) {
  int t := A;
  int c := 1;
  while (c = 1) {
    if (t = A) {
      (c := 0 [a] t := B);
    } else {
      (c := 0 [b] t := A);
    }
  }
  return t;
}
```

Aim: find expectation $\mathcal{T}$

Satisfying $\mathcal{T} \leqslant [\mathrm{t} = A]$ upon termination.

Observation

On entering the loop, $\mathrm{c} = 1$ and either $\mathrm{t} = A$ or $\mathrm{t} = B$.

Template suggestion

$$\mathcal{T} \quad = \quad \underbrace{[\mathrm{t} = A \wedge \mathrm{c} = 0]}_{A \text{ wins duel}} \cdot 1 + \underbrace{[\mathrm{t} = A \wedge \mathrm{c} = 1]}_{A\text{'s turn}} \cdot \alpha + \underbrace{[\mathrm{t} = B \wedge \mathrm{c} = 1]}_{B\text{'s turn}} \cdot \beta$$

Duelling cowboys: when does A win?

Invariant template

$\mathcal{T} = [\mathrm{t} = A \wedge \mathrm{c} = 0] \cdot 1 + [\mathrm{t} = A \wedge \mathrm{c} = 1] \cdot \alpha + [\mathrm{t} = B \wedge \mathrm{c} = 1] \cdot \beta$

Initially, $\mathrm{t} = A \wedge \mathrm{c} = 1$ and thus $\alpha = Pr\{A \text{ wins duel}\}$.

Running PrinSys yields

$a \cdot \beta - a + \alpha - \beta \leqslant 0 \quad \wedge \quad b \cdot \alpha - \alpha + \beta \leqslant 0$

Simplification yields

$\beta \leqslant (1-b) \cdot \alpha$ and $\alpha \leqslant \dfrac{a}{a + b - a \cdot b}$

As we want to maximise the probability to win

$\beta = (1-b) \cdot \alpha$ and $\alpha = \dfrac{a}{a + b - a \cdot b}$

It follows that cowboy A wins the duel with probability $\frac{a}{a+b-a \cdot b}$.

Quantitative loop invariant

Annotated program for post-expectation $[t = A]$

```
int cowboyDuel(a, b) {
```

$$\langle \tfrac{(1-b)a}{a+b-ab} \rangle$$

$$\langle \min\{\tfrac{a}{a+b-ab}, \tfrac{(1-b)a}{a+b-ab}\} \rangle$$

```
  (t := A [] t := B);
```

$$\langle [t = A] \cdot \tfrac{a}{a+b-ab} + [t = B] \cdot \tfrac{(1-b)a}{a+b-ab} \rangle$$

```
  c := 1;
```

$$\langle [t = A \wedge c = 0] \cdot 1 + [t = A \wedge c = 1] \cdot \tfrac{a}{a+b-ab} + [t = B \wedge c = 1] \cdot \tfrac{(1-b)a}{a+b-ab} \rangle$$

```
  while (c = 1) {
```

$$\langle [t = A \wedge c = 1] \cdot \tfrac{a}{a+b-ab} + [t = B \wedge c = 1] \cdot \tfrac{(1-b)a}{a+b-ab} \rangle$$

$$\langle [t = A \wedge c \neq 1] \cdot a + [t = A \wedge c = 1] \cdot \tfrac{a}{a+b-ab} + [t = B \wedge c = 0] \cdot (1-b) + [t = B \wedge c = 1] \cdot \tfrac{(1-b)a}{a+b-ab} \rangle$$

```
    if (t = A) {
      (c := 0 [a] t := B);
    } else {
      (c := 0 [b] t := A);
    }
```

$$\langle [t = A \wedge c = 0] \cdot 1 + [t = A \wedge c = 1] \cdot \tfrac{a}{a+b-ab} + [t = B \wedge c = 1] \cdot \tfrac{(1-b)a}{a+b-ab} \rangle$$

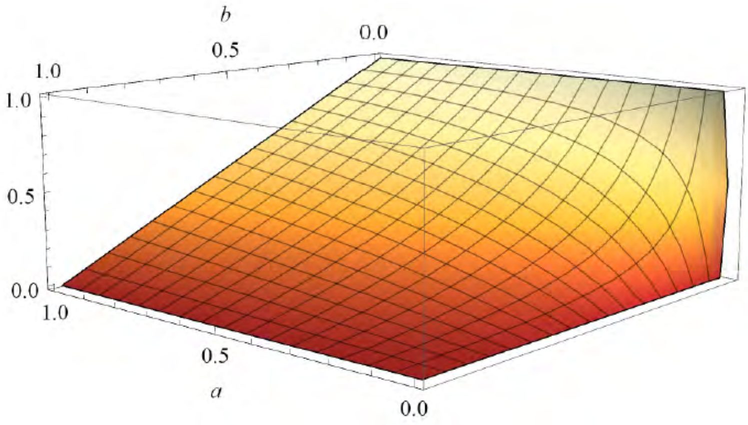
```
  }
```

$$\langle [c \neq 1] \cdot \Big([t = A \wedge c = 0] \cdot 1 + [t = A \wedge c = 1] \cdot \tfrac{a}{a+b-ab} + [t = B \wedge c = 1] \cdot \tfrac{(1-b)a}{a+b-ab} \Big) \rangle$$

$$\langle [t = A] \rangle$$

```
  return t; // the survivor
}
```

When one starts nondeterministically



Cowboy A wins the duel with probability at least $\dfrac{(1-b)\cdot a}{a+b-a\cdot b}$.

Program equivalence

```
int XminY1(float p, q){
  int  x, f := 0, 0;
  while (f = 0) {
    (x +:= 1 [p] f := 1);
  }
  f := 0;
  while (f = 0) {
    (x −:= 1 [q] f := 1);
  }
  return x;
}
```

```
int XminY2(float p, q){
  int x, f := 0, 0;
  (f := 0 [0.5] f := 1);
  if (f = 0) {
    while (f = 0) {
      (x +:= 1 [p] f := 1);
    }
  } else {
    f := 0;
    while (f = 0) {
      x −:= 1;
      (skip [q] f := 1);
    }
  }
return x;
}
```

Using template $\mathcal{T} = x + [\mathsf{f} = 0] \cdot \alpha$ we find the invariants :

$\alpha_{11} = \frac{p}{1-p}$, $\alpha_{12} = -\frac{q}{1-q}$, $\alpha_{21} = \alpha_{11}$ and $\alpha_{22} = -\frac{1}{1-q}$.

Overview

Recursive probabilistic programs

Probabilistic pushdown automata **[Esparza *et al.*, 2004]**

Are a natural model for recursive probabilistic programs.

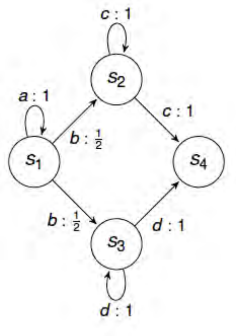
Verification results for fully probabilistic PDA

- pPDA are equally expressive as recursive Markov chains [E & Y 2009]
- Qualitative pCTL model checking is decidable [Esparza *et al.*, 2004]
- Quantitative checking of ω-regular properties is decidable [same]
- Expectations (time, long run) are effectively computable [same, +1]

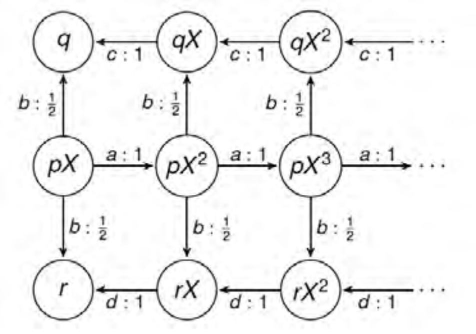
When does a recursive program meet a finite specification?

Example

Specification: pTS



Implementation: example pPDA



Claim: $\mathbf{pX^{n+1} \sqsubseteq s_1}$

Proof: $\{ (pX^{n+1}, s_1), (qX^n, s_2), (rX^n, s_3) \mid n > 0 \} \cup \{ (q, s_4), (r, s_4) \}$ is a probabilistic simulation.

Recursive probabilistic programs

Complexity results

	(coupled) bisimilarity	(coupled) similarity
PDA vs. finite TS	PSPACE-complete	EXPTIME-complete
pPDA vs. finite pTS	EXPTIME-complete	EXPTIME-complete[2]

[2] This also applies to checking simulation equivalence.

Epilogue

Take-home message

- Connection between wp-semantics and operational semantics.
- Synthesizing probabilistic loop invariants using constraint solving.

$\Rightarrow$ First step towards automated probabilistic program analysis.

- Prototypical tool-support PRINSYS.

Future work

- Observe statements.
- Program minimization.
- Average time-complexity analysis.

Fin.

Further reading

- B. Kaminski and J.-P. K.
  Analyzing expected outcomes and almost-sure termination is hard.
  2014 (unpublished).
- F. Gretz, J.-P. K., and A. McIver.
  Operational versus wp-semantics for `pGCL`*.*
  Performance Evaluation, 2014.
- F. Gretz, J.-P. K., and A. McIver.
  PRINSYS — on a quest for probabilistic loop invariants.
  QEST 2013.
- H. Fu, and J.-P. K..
  Deciding probabilistic simulation between probabilistic pushdown automata and finite-state systems. FSTTCS 2011.
- J.-P. K., A. McIver, L. Meinicke, and C. Morgan.
  Linear-invariant generation for probabilistic programs.
  SAS 2010.