Once upon a time .......

Duelling cowboys

```c
int cowboyDuel(float a, b) { // 0 < a < 1, 0 < b < 1
    int t := A; // cowboy A has first shooting turn
    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; // who survived the duel?
}
```

Claim:
Cowboy A wins the duel with probability $\frac{a}{a+b-a \cdot b}$. 
Geometric distribution

Let $X$ be a discrete random variable, natural $x > 0$ and real $0 < p \leq 1$. The probability that the $x$-th trial (out of $x$ trials) is the first success is:

$$Pr\{X = x\} = (1 - p)^{x-1} \cdot p$$

Playing with geometric distributions [Kiefer et. al., 2012]

- $X$ 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 $X - Y$

```c
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

```c
int XminY2(float p, q){
    int x := 0;
    bool flip := false;
    (flip := false [0.5] flip := true); // flip a fair coin
    if (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 [q] flip := true);
        }
    }
    return x; // a sample of X-Y
}
```

Program equivalence

```c
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 [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?
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
  ⇒ 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]

- Use Floyd-Hoare style reasoning for probabilistic programs.
  ⇒ allowing for backward post-pre-condition reasoning.

Program equivalence

Our analysis yields:
Both programs are equivalent for any \( q \) with \( q = \frac{1}{2} - \frac{p}{2} \).

Duelling cowboys

Cowboy A wins the duel with probability \( \frac{a}{a + b - ab} \).
Roadmap of the talk

1. Introduction
2. Probabilistic guarded command language
3. Operational semantics of pGCL
4. Denotational semantics of pGCL
5. Denotational vs. operational semantics of pGCL
6. Synthesizing loop invariants
7. Epilogue

Overview

1. Introduction
2. Probabilistic guarded command language
3. Operational semantics of pGCL
4. Denotational semantics of pGCL
5. Denotational vs. operational semantics of pGCL
6. Synthesizing loop invariants
7. Epilogue

Dijkstra’s guarded command language

- skip
- abort
- x := E
- prog1 ; prog2
- if (G) prog1 else prog2
- prog1 [] prog2
- while (G) prog

Probabilistic guarded command language

- skip
- abort
- x := E
- prog1 ; prog2
- if (G) prog1 else prog2
- prog1 [] prog2
- prog1 [p] prog2
- while (G) prog
Overview

1. Introduction
2. Probabilistic guarded command language
3. Operational semantics of pGCL
4. Denotational semantics of pGCL
5. Denotational vs. operational semantics of pGCL
6. Synthesizing loop invariants
7. Epilogue

Markov decision processes

Markov decision process

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

- $S$ is a countable set of states with initial state-set $S_0 \subseteq S$, $S_0 \neq \emptyset$
- $\rightarrow \subseteq S \times \text{Dist}(S)$ is a transition relation

Notation:

$s \rightarrow \mu$ denotes $(s, \mu) \in \rightarrow$ and $s \rightarrow t$ denotes $s \rightarrow \mu$ with $\mu(t) = 1$. Let $\text{Dist}(s) = \{ \mu \mid s \rightarrow \mu \}$. Distributions may be symbolic, e.g., $\mu(s) = p$.

Intuitive operational behavior

1. Nondeterministically select some initial state $s_0 \in S_0$
2. In state $s$ with $\text{Dist}(s) \neq \emptyset$, nondeterministically select $\mu \in \text{Dist}(s)$
3. The next state $t$ is randomly chosen with probability $\mu(t)$.
4. If $\text{Dist}(t) = \emptyset$, exit; otherwise go back to step 2.

Policies

Reasoning about probabilities of sets of paths of an MDP relies on the resolution of nondeterminism. This resolution is performed by a policy.

Policy

Function $\Psi : S \rightarrow \text{Dist}(S)$ is a positional policy for MDP $\mathcal{M} = (S, S_0, \rightarrow)$ with $\Psi(s) \in \text{Dist}(s)$ for all $s \in S$.

Alternating sequence

$$\pi = s_0 \overset{\mu_0}{\rightarrow} s_1 \overset{\mu_1}{\rightarrow} \ldots$$

is a path of $\mathcal{M}$ whenever $\mu_i(s_{i+1}) > 0$ for all $i \geq 0$.

It is called a $\Psi$-path if $\Psi(s_{i-1}) = \mu_i$ for all $i \geq 0$. Let $\text{Paths}^\Psi(s)$ denote the set of $\Psi$-paths starting from state $s$.

1Also called scheduler, strategy or adversary.
**Induced Markov chain**

Each (positional) policy induces a Markov chain.

**Operational semantics of pGCL**

**Aim:** Model the behaviour of a program $P \in pGCL$ by an MDP $M[P]$.

**Approach:**
- Let $\eta$ 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 $Q \in pGCL$ or $Q = \text{exit}$
- Initial states are tuples $\langle P, \eta \rangle$ where $\eta$ fulfils the initial conditions
- Transition relation $\rightarrow$ is the smallest relation satisfying the inference rules (cf. next slide)

**Structured operational semantics**

$$
\langle \text{skip}, \eta \rangle \rightarrow \langle \text{exit}, \eta \rangle \quad \langle \text{abort}, \eta \rangle \rightarrow \langle \text{abort}, \eta \rangle
$$

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

$$
\langle P[Q], \eta \rangle \rightarrow \langle P, \eta \rangle \quad \langle P[Q], \eta \rangle \rightarrow \langle Q, \eta \rangle
$$

$$
\begin{align*}
\langle P Q, \eta \rangle & \rightarrow \mu \text{ with } \mu(\langle P, \eta \rangle) = \rho \text{ and } \mu(\langle Q, \eta \rangle) = 1 - \rho \\
\langle P, \eta \rangle & \rightarrow \nu \text{ with } \nu(\langle P', Q', \eta' \rangle) = \mu(\langle P', \eta' \rangle) \text{ where exit; } P = P
\end{align*}
$$

$$
\begin{align*}
\eta \models G & \rightarrow \langle P; \text{while } G \{P \}, \eta \rangle \\
\eta \not\models G & \rightarrow \langle \text{exit}, \eta \rangle
\end{align*}
$$

**MDP of duelling cowboys**

```java
int cowboyDuel(float a, b) {
    int t := A;
    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

Introduction

Probabilistic guarded command language

Operational semantics of pGCL

Denotational semantics of pGCL

Denotational vs. operational semantics of pGCL

Synthesizing loop invariants

Epilogue

Weakest preconditions

Weakest pre-condition

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" pre-condition $E$ 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 $E \Rightarrow wp(P, F)$.

Weakest liberal pre-condition

A weakest liberal pre-condition $wlp(P, F)$ yields the weakest pre-condition for which $P$ either does not terminate or establishes $F$. It does not ensure termination and corresponds to Hoare logic in partial correctness.

Expectations

Weakest pre-expectation

An expectation is an integer-valued function on the program variables. 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 expected value $e$ on $P$’s initial state ensuring that $P$’s execution terminates with a value $f$.

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

Weakest liberal pre-expectation

A weakest liberal pre-expectation $wlp(P, f)$ yields the least expectation for which $P$ either does not terminate or establishes $f$. 

Predicate transformer semantics of Dijkstra’s GCL

Syntax

Semantics $wp(P, F)$

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\triangleright$ skip</td>
<td>$F$</td>
</tr>
<tr>
<td>$\triangleright$ abort</td>
<td>$false$</td>
</tr>
<tr>
<td>$\triangleright$ x := E</td>
<td>$F[x := E]$</td>
</tr>
<tr>
<td>$\triangleright$ P1 ; P2</td>
<td>$wp(P_1, wp(P_2, F))$</td>
</tr>
<tr>
<td>$\triangleright$ if (G)P1 else P2</td>
<td>$(G \Rightarrow wp(P_1, F)) \land (\neg G \Rightarrow wp(P_2, F))$</td>
</tr>
<tr>
<td>$\triangleright$ P1 [] P2</td>
<td>$wp(P_1, F) \land wp(P_2, F)$</td>
</tr>
<tr>
<td>$\triangleright$ while (G)P</td>
<td>$\mu X. ((G \Rightarrow wp(P, X)) \land (\neg G \Rightarrow F))$</td>
</tr>
</tbody>
</table>

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

$wlp$-semantics differs from $wp$-semantics only for $while$. 

Weakest pre-expectation

A weakest pre-expectation $wp(P, f)$ yields the least expected value $e$ on $P$’s initial state ensuring that $P$’s execution terminates with a value $f$.

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

Weakest liberal pre-expectation

A weakest liberal pre-expectation $wlp(P, f)$ yields the least expectation for which $P$ either does not terminate or establishes $f$. 

Weakest pre-condition

Weakest precondition

[Dijsktra 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" pre-condition $E$ 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 $E \Rightarrow wp(P, F)$.

Weakest liberal precondition

A weakest liberal pre-condition $wlp(P, F)$ yields the weakest pre-condition for which $P$ either does not terminate or establishes $F$. It does not ensure termination and corresponds to Hoare logic in partial correctness.
## Expectation transformer semantics of \( pGCL \)

### Syntax

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Semantics ( wp(P, f) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>▶ skip</td>
<td>▶ ( f )</td>
</tr>
<tr>
<td>▶ abort</td>
<td>▶ 0</td>
</tr>
<tr>
<td>▶ ( x := E )</td>
<td>▶ ( f[x := E] )</td>
</tr>
<tr>
<td>▶ ( P_1 ; P_2 )</td>
<td>▶ ( wp(P_1, wp(P_2, f)) )</td>
</tr>
<tr>
<td>▶ ( \text{if } (G) P_1 \text{ else } P_2 )</td>
<td>▶ ( [G] \cdot wp(P_1, f) + [-G] \cdot wp(P_2, f) )</td>
</tr>
<tr>
<td>▶ ( P_1 [p] P_2 )</td>
<td>▶ ( \min(wp(P_1, f), wp(P_2, f)) )</td>
</tr>
<tr>
<td>▶ ( \text{while } (G) P )</td>
<td>▶ ( p \cdot wp(P, f) + (1-p) \cdot wp(P_2, f) )</td>
</tr>
<tr>
<td></td>
<td>▶ ( \mu X. ([G] \cdot wp(P, X) + [-G] \cdot f) )</td>
</tr>
</tbody>
</table>

\( \mu \) is the least fixed point operator wrt. the ordering \( \leq \) on expectations.

wlp-semantics differs from wp-semantics only for \( \text{while} \) and \( \text{abort} \).

### Example weakest pre-expectations

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

- If \( f = \left[ all(\heartsuit) \right] \), then \( wlp(flip, f) = \frac{1}{8} \).
- If \( g = 10 \cdot [all(\heartsuit)] + 5 \cdot [all(\heartsuit)] \), then:
  \[
  wlp(flip, g) = 6 \cdot \frac{1}{8} \cdot 0 + 1 \cdot \frac{1}{8} \cdot 10 + 1 \cdot \frac{1}{8} \cdot 5 = \frac{15}{8}
  \]

## A simple slot machine

```java
void flip {
    d1 := ♥ [1/2] ◊;
    d2 := ♥ [1/2] ◊;
    d3 := ♥ [1/2] ◊;
}
```

## 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 \((M, r)\) with \( M \) an MDP with state space \( S \) and \( r : S \rightarrow \mathbb{Z} \) a function assigning an integer reward to each state.

Intuitively, the reward \( r(s) \) stands for the reward earned on leaving state \( s \).

### Cumulative cost for reachability

Let \( \pi = s_0 \xrightarrow{r_{s_0}} s_1 \xrightarrow{r_{s_1}} \ldots \) be an infinite path in \((M, r)\) and \( T \subseteq S \) a set of target states such that \( \pi \models \Diamond T \). The cumulative cost along \( \pi \) 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 \( r_T(\pi) = 0 \).
Cost-bounded reachability

Expected reward for reachability

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

$$\text{ExpRew}(s \models \Diamond T) = \min_{\Psi} \sum_{c=-\infty}^{\infty} c \cdot \Pr_{\Psi}(s \models \Diamond T | r_T(\pi) = c)$$

A demonic positional policy corresponds to a weakest pre-expectation.

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

$$\text{CExpRew}(s \models \Diamond T) = \min_{\Psi} \sum_{c=-\infty}^{\infty} c \cdot \Pr_{\Psi}(s \models \Diamond T | r_T(\pi) = c)$$

where $\Psi$ is a policy such that $\Pr_{\Psi}(s \models \Diamond T) > 0$.

$\text{CExpRew}$ is thus the minimal expected reward under the condition $\Diamond T$.

Relating operational and wp-semantics of pGCL

Weakest pre-expectations vs. expected reachability rewards

For pGCL-program $P$, variable valuation $\eta$, and post-expectation $f$:

$$\wp(P, f)(\eta) = \text{ExpRew}^{M}[P](\langle P, \eta \rangle \models \Diamond \{\langle \text{exit}, \eta' \rangle \})$$

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

$$\wlp(P, f)(\eta) = \text{CExpRew}^{M}[P](\langle P, \eta \rangle \models \Diamond \{\langle \text{exit}, \eta' \rangle \})$$

Thus, $\wp(P, f)$ evaluated at $\eta$ is the minimal expected value of $f$ over any of the result distributions of $P$. The weakest liberal pre-expectation $\wp(P, f)$ is similar under the condition that the program terminates.

Expected rewards in finite MDPs without parameters can simply be computed by solving an LP problem.

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:

$$G \wedge I \Rightarrow \wp(P, I)$$ (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 program whenever 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:

   \[ \alpha_1 \cdot x_1 + \ldots + \alpha_n \cdot x_n + \alpha_{n+1} < 0 \]

   where \( \alpha_i \) is a parameter and \( x_i \) a program variable.

2. Express verification conditions as inequality constraints over \( \alpha_i, x_i \).

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 invariant

Recall that for while-loops we have:

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

To determine this wp, we use an “invariant” \( I \) such that \( \neg G \cdot I \leq f \).

Quantitative loop invariant

Expectation \( I \) is a quantitative loop invariant if —by consecution—

- it is preserved by loop iterations: \( [G] \cdot I \leq \text{wp}(P, I) \).

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 \( \alpha_i, \beta_i \), program variable \( x_i \), and \( \ll \in \{<, \leq\} \).

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.
Disjoint normal form

For any linear loop-free program \( P \in \text{pGCL} \), and linear expression \( f \), \( \wp(P, f) \) is expressible as a linear expression.

Disjoint normal form

A quantitative expression of the form \( \sum_{0 < m \leq M} [\land_{0 < n \leq N} P_{mn}] \cdot Q_m \) is in disjoint normal form (DNF) if for all \( 0 < i, j \leq M \) where \( i \neq j \), we have:

\[
\land_{0 < m \leq N} P_{in} \land \land_{0 < n \leq N} P_{jn} = \text{false}
\]

Obtaining existentially quantified FO-formulas

Motzkin’s transposition theorem is one of the deepest results in the part of mathematics dealing with linear inequalities

[Emilie Delabie, Encyclopedia of Optimization, 2009]

From numerical constraints to predicates

Let \( Q_{MN} \) be a linear expression with equivalent DNF linear expression

\[
[P_1] \cdot Q_1 + \ldots + [P_M] \cdot Q_M
\]

and \( Q'_{KL} \) be a linear expression with equivalent DNF linear expression

\[
[P'_1] \cdot Q'_1 + \ldots + [P'_K] \cdot Q'_K.
\]

Then:

\[
Q_{MN} \leq Q'_{KL}
\]

if and only if for all \( m \in [1..M] \), and \( n \in [1..K] \):

\[
P_m \land P'_{n} \Rightarrow (Q_m - Q'_n \leq 0)
\]

\[
P_m \land (\land_{k \in [1..K]} \neg P'_k) \Rightarrow Q_m \leq 0
\]

Any inequality \( Q \leq Q' \) between propositionally linear expressions is equivalent to a finite Boolean expression over linear constraints.

Motzkin’s transposition theorem (1936)

Let \( A, A' \) be matrices, \( b, b' \) column vectors, and \( x \) a column vector of variables. If \( A \cdot x \leq b \) and \( A' \cdot x < b' \) is unsatisfiable, then there exist row vectors \( \lambda, \lambda' \) with:

\[
\lambda \geq 0 \text{ and } \lambda' \geq 0 \text{ and } \lambda \cdot A + \lambda' \cdot A' = 0
\]

and either

1. \( \lambda \cdot b + \lambda' \cdot b' > 0 \), or
2. some entry of \( \lambda' \) is strictly positive and \( \lambda \cdot b + \lambda' \cdot b' \geq 0 \).

(\( \lambda \) and \( \lambda' \) form a witness of \( A \cdot x \geq b \) and \( A' \cdot x > b' \) being unsatisfiable.)
**Prinsys Tool:** Synthesis of Probabilistic Invariants

![Diagram of the Prinsys tool process]

Download from moves.rwth-aachen.de/prinsys

---

**Duelling cowboys: when does A win?**

**Invariant template**

\[ T = \{ t = A \land c = 0 \} \cdot 1 + \{ t = A \land c = 1 \} \cdot a + \{ t = B \land c = 1 \} \cdot \beta \]

Initially, \( t = A \land c = 1 \) and thus \( \alpha = Pr\{A \text{ wins duel}\} \).

**Running PrinSys yields**

\[ a \cdot \beta - a + \alpha - \beta \leq 0 \land b \cdot \alpha - \alpha + \beta \leq 0 \]

**Simplification yields**

\[ \beta \leq (1 - b) \cdot \alpha \quad \text{and} \quad \alpha \leq \frac{a}{a + b - a \cdot b} \]

As we want to maximise the probability to win

\[ \beta = (1 - b) \cdot \alpha \quad \text{and} \quad \alpha = \frac{a}{a + b - a \cdot b} \]

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

**Quantitative loop invariant**

\[ T = \{ t = A \land c = 0 \} \cdot 1 + \{ t = A \land c = 1 \} \cdot a + \{ t = B \land c = 1 \} \cdot (1 - b) \cdot \alpha \]

---

**Program equivalence**

**Template suggestion**

\[ T = \{ t = A \land c = 0 \} \cdot 1 + \{ t = A \land c = 1 \} \cdot a + \{ t = B \land c = 1 \} \cdot \beta \]

**Expected value of** \( x \) **is**

\[ p \cdot (1 - p) - q \cdot (1 - q) \]

Using template \( T = x + [f = 0] \cdot \alpha \) we find the invariants:

\[ \alpha_{11} = \frac{p}{1 - p}, \alpha_{12} = \frac{q}{1 - q}, \alpha_{21} = \alpha_{11} \quad \text{and} \quad \alpha_{22} = -\frac{1}{1 - q} \]
Recursive probabilistic programs

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

Are a natural model for recursive probabilistic programs. Checking whether they simulate (or are simulated by) a finite Markov chain is EXPTIME-complete.

### Overview of complexities

<table>
<thead>
<tr>
<th>(combined) bisimilarity</th>
<th>(combined) similarity</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDA vs. finite TS</td>
<td>PSPACE-complete</td>
</tr>
<tr>
<td></td>
<td>EXPTIME-complete</td>
</tr>
<tr>
<td>pPDA vs. finite pTS</td>
<td>EXPTIME-complete</td>
</tr>
</tbody>
</table>

---

**Epilogue**

**Take-home message**

- Connection between wp-semantics and operational semantics.
- Synthesizing probabilistic loop invariants using constraint solving.
- Large potential for automated probabilistic program analysis.
- Initial prototypical tool-support Prinsys is available.

**Future work**

- Further development of Prinsys.
- Non-linear probabilistic programs.
- Average time-complexity analysis.

**Congratulations**