Automated Analysis of Probabilistic Programs

Joost-Pieter Katoen

Software Modeling and Verification Group
RWTH Aachen University

joint work with Friedrich Gretz and Annabelle McIver

June 12, 2013
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
Probabilistic programs

What are probabilistic programs?
Sequential, possibly non-deterministic, programs with random assignments.

Applications
Cryptography, privacy, quantum computing, and 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.
⇒ Our aim: push the limits of automated analysis

\(^1\)Their analysis is undecidable.
Example: Probabilistic encryption

Goldwasser and Micali proved (in 1982) that encryption schemes must be randomized rather than deterministic [...] a development that revolutionized the study of cryptography and laid the foundation for the theory of cryptographic security.

Used in almost all communications protocols, Internet transactions and cloud computing.
The famous RSA-OAEP protocol

**Oracle** \( \text{Enc}_{pk}(m) \):
- \( r \triangleq \{0, 1\}^{k_0} \);
- \( s \leftarrow G(r) \oplus (m \| 0^{k_1}) \);
- \( t \leftarrow H(s) \oplus r \);
- return \( f_{pk}(s \| t) \)

**Oracle** \( \text{Dec}_{sk}(c) \):
- \( (s, t) \leftarrow f^{-1}(c) \);
- \( r \leftarrow t \oplus H(s) \);
- if \( [s \oplus G(r)]_{k_1} = 0^{k_1} \) then return \( [s \oplus G(r)]^n \)
- else return \( \bot \)

**Oracle** \( G(x) \):
- if \( x \notin \text{dom}(L_G) \) then \( L_G[x] \triangleq \{0, 1\}^{n+k_1} \);
- return \( L_G[x] \)

**Oracle** \( H(x) \):
- if \( x \notin \text{dom}(L_H) \) then \( L_H[x] \triangleq \{0, 1\}^{k_0} \);
- return \( L_H[x] \)

**Game** IND-CCA2 :
- \( (sk, pk) \leftarrow \mathcal{K}_G() \);
- \( (m_0, m_1, \sigma) \leftarrow \mathcal{A}_1(pk) \);
- \( b \triangleq \{0, 1\} \);
- \( c^* \leftarrow \text{Enc}(pk, m_b) \);
- \( b' \leftarrow \mathcal{A}_2(pk, c^*, \sigma) \);
- return \( b = b' \)

**Game** POW :
- \( (sk, pk) \leftarrow \mathcal{K}_G() \);
- \( y \triangleq \{0, 1\}^{n+k_1} \);
- \( z \triangleq \{0, 1\}^{k_0} \);
- \( y' \leftarrow \mathcal{I}(f_{pk}(y \| z)) \);
- return \( y = y' \)
Its proof took a very long time

1994 Purported proof of chosen-ciphertext security
2001 Proof establishes a weaker security notion, but desired security can be achieved
   1. ...for a modified scheme, or
   2. ...under stronger assumptions
2004 Filled gaps in Fujisaki et al. 2001 proof
2009 Security definition needs to be clarified
2010 Filled gaps and improved bounds from 2004 proof
2012 Improved bound from 2010 proof

[Gilles Barthe: Computer-aided Cryptographic Proofs, keynote talk ETAPS’13]
State of the art

Automated probabilistic program analysis is at its infancy.
Once upon a time .......
Duelling cowboys

```c
int cowboyDuel(float a, b) { // 0 < a < 1, 0 < b < 1
    int t := A []; t := B; // decide cowboy for first shooting
    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 $X - Y$

```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 (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

```c
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;
}
```

```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 \((p, q) = \left(\frac{1}{2}, \frac{2}{3}\right)\).

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

Apply deductive verification!

[McIver & Morgan]
Duelling cowboys

```c
int cowboyDuel(float a, b) {  // 0 < a < 1, 0 < b < 1
    int t := A [] t := B;  // decide which cowboy 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;  // the survivor
}
```

We can infer:

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

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**

**empty statement**

**abortion**

**assignment**

**sequential composition**

**choice**

**non-deterministic choice**

**iteration**
Probabilistic guarded command language \( pGCL \)

- **skip**
- **abort**
- **\( x := E \)**
- **\( \text{prog}_1 ; \text{prog}_2 \)**
- **\( \text{if } (G) \text{ prog}_1 \text{ else } \text{prog}_2 \)**
- **\( \text{prog}_1 [] \text{ prog}_2 \)**
- **\( \text{prog}_1 [p] \text{ prog}_2 \)**
- **\( \text{while } (G) \text{ prog} \)**
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

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:
Let $\text{Dist}(s) = \{ \mu \mid (s, \mu) \in \rightarrow \}$.
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. \footnote{Also called scheduler, strategy or adversary.}

Policy

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

Alternating sequence

$$\pi = s_0 \xrightarrow{\mu_0} s_1 \xrightarrow{\mu_1} \ldots$$

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

It is called a \textit{$\mathcal{P}$-path} if $\mathcal{P}(s_{i-1}) = \mu_i$ for all $i \geq 0$.

Let $\text{Paths}^{\mathcal{P}}(s)$ denote the set of $\mathcal{P}$-paths starting from state $s$. 
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
### MDP of duelling cowboys

```c
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

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
Weakest preconditions

Weakest precondition

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 $E \Rightarrow wp(P, F)$.

Weakest liberal precondition

A weakest liberal precondition $wlp(P, F)$ yields the weakest precondition for which $P$ either does not terminate or establishes $F$. It does not ensure termination and corresponds to Hoare logic in partial correctness.
### Predicate transformer semantics of Dijkstra’s GCL

#### Syntax

- `skip`
- `abort`
- `x := E`
- `P1 ; P2`
- `if (G)P1 else P2`
- `P1 [] P2`
- `while (G)P`

#### Semantics \( wp(P, F) \)

- `F`
- `false`
- `F[x := E]`
- `wp(P_1, wp(P_2, F))`
- `(G \Rightarrow wp(P_1, F)) \land (\neg G \Rightarrow wp(P_2, F))`
- `wp(P_1, F) \land wp(P_2, F)`
- `\mu X. ((G \Rightarrow wp(P, X)) \land (\neg G \Rightarrow F))`

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

wlp-semantics differs from wp-semantics only for `while` and `abort`. 
Expectations

Weakest pre-expectation

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 \( \text{wp}(P, f) \) 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 \( e \leq \text{wp}(P, f) \), where \( \leq \) is to be interpreted in a point-wise manner.

Weakest liberal pre-expectation

A weakest liberal pre-expectation \( \text{wlp}(P, f) \) yields the least expectation for which \( P \) either does not terminate or establishes \( f \).
### Expectation transformer semantics of \( \text{pGCL} \)

#### Syntax

- `skip`
- `abort`
- `x := E`
- `P1 ; P2`
- `if (G)P1 else P2`
- `P1 [\ p\ ] P2`
- `P1 [p] P2`
- `while (G)P`

#### Semantics \( wp(P, f) \)

- \( f \)
- \( 0 \)
- \( f[x := E] \)
- \( wp(P_1, wp(P_2, f)) \)
- \( [G] \cdot wp(P_1, f) + [\neg G] \cdot wp(P_2, f) \)
- \( \min (wp(P_1, f), wp(P_2, f)) \)
- \( p \cdot wp(P_1, f) + (1-p) \cdot wp(P_2, f) \)
- \( \mu X. ([G] \cdot wp(P, X) + [\neg G] \cdot f) \)

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

\( wlp \)-semantics differs from \( wp \)-semantics only for `while` and `abort`. 

---

Joost-Pieter Katoen
A simple slot machine

```c
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 \( f = [all(♥)] \), then \( wlp(flip, f) = \frac{1}{8} \).
- If \( g = 10 \cdot [all(♥)] + 5 \cdot [all(♦)] \), then:

\[
\begin{align*}
    wlp(flip, g) &= 6 \cdot \frac{1}{8} \cdot 0 + 1 \cdot \frac{1}{8} \cdot 10 + 1 \cdot \frac{1}{8} \cdot 5 \\
    &= \frac{15}{8}
\end{align*}
\]

So the least fraction of the jackpot the gamer can expect to win is \( \frac{15}{8} \).
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
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 \(r : S \to \mathbb{Z}\) a function assigning an integer reward to each state.

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

## Cumulative cost for reachability

Let \(\pi = s_0 \xrightarrow{\mu_0} s_1 \xrightarrow{\mu_1} \ldots\) 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:

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

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:

$$ERew(s \models \diamond T) = \min \sum_{c=0}^{\infty} c \cdot Pr^{\mathfrak{P}} \{ \pi \in \text{Paths}^{\mathfrak{P}}(s, \diamond T) \mid r_T(\pi) = c \}$$

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

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

$$LERew(s \models \diamond T) = \min \left\{ \sum_{c=0}^{\infty} c \cdot Pr^{\mathfrak{P}} \{ \pi \in \text{Paths}^{\mathfrak{P}}(s, \diamond T) \mid r_T(\pi) = c \} \right\} + Pr^{\mathfrak{P}}(s \not\models \diamond T)$$

$LExpRew$ is the minimal expected reward under $\diamond T$ or never reach $T$. 

[Joost-Pieter Katoen] 35/51
Relating operational and wp-semantics of pGCL

Weakest pre-expectations vs. expected reachability rewards

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

$$wp(P, f)(\eta) = ERew^M[\llbracket P \rrbracket](\langle P, \eta \rangle \models \lozenge P^\top)$$

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

$$wlwp(P, f)(\eta) = LERew^M[\llbracket P \rrbracket](\langle P, \eta \rangle \models \lozenge P^\top)$$

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.
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
Qualitative loop invariants

Recall that for while-loops we have:

\[ \wp(\text{while}(G)\{P\}, F) = \mu X. (G \Rightarrow \wp(P, X) \land \neg G \Rightarrow F) \]

To determine this \( \wp \), one exploits an “invariant” \( I \) such that \( \neg G \land I \Rightarrow F \).

Loop invariant

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

\[ G \land I \Rightarrow \wp(P, I) \]  \hspace{1cm} (consecution condition)

Then: \( \{ I \} \ \text{while}(G)\{P\} \{ F \} \) 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:

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

   where \( \alpha_j \) 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 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 $[\neg G] \cdot I \leq f$.

Quantitative loop invariant

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

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

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.
Example: run the slot machine repeatedly

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

```cpp
void playGame {
    flip; // init
    while ¬(all(♥) ∨ all(♦)) {
        flip;
    }
}
```

Example loop invariant

Let post-expectation \( f = 1 \cdot [all(♥)] + \frac{1}{2} \cdot [all(♦)] \)

- Invariant \( I = \frac{3}{4} \cdot [¬all(♥) ∧ ¬all(♦)] + 1 \cdot [all(♥)] + \frac{1}{2} \cdot [all(♦)]. \)
- As \( f = [all(♥) ∨ all(♦)] \cdot I \) we have \( \{ I \} \text{ loop } \{ f \} \)
- It follows \( wlp(flip, I) = \frac{3}{4} \).
- In 50% the loop terminates with all ♥, in 50% with all ♦.
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.
Soundness and completeness

**Theorem**

For any linear $\text{pGCL}$ program annotated with propositionally linear expressions, our method can be used to 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?

**Aim: find expectation** $\mathcal{T}$

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

**Observation**

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

**Template suggestion**

$$
\mathcal{T} = \underbrace{[t = A \land c = 0]}_{A \text{ wins duel}} \cdot 1 \\
+ \underbrace{[t = A \land c = 1]}_{A\text{'s turn}} \cdot \alpha \\
+ \underbrace{[t = B \land c = 1]}_{B\text{'s turn}} \cdot \beta
$$
### Duelling cowboys: when does A win?

#### Invariant template

\[
T = [t = A \wedge c = 0] \cdot 1 + [t = A \wedge c = 1] \cdot \alpha + [t = B \wedge c = 1] \cdot \beta
\]

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

#### Running \textsc{PrinSys} yields

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

#### 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 \( \frac{a}{a + b - a \cdot b} \).
Annotated program for post-expectation \([ t = A ]\)

```plaintext
int cowboyDuel(a, b) {
    \( \langle \frac{(1-b)a}{a+b-ab} \rangle \)
    \( \langle \min\{\frac{a}{a+b-ab}, \frac{(1-b)a}{a+b-ab}\} \rangle \)
    (t := A \ [ ] t := B);
    \( \langle [t = A] \cdot \frac{a}{a+b-ab} + [t = B] \cdot \frac{(1-b)a}{a+b-ab} \rangle \)
    c := 1;
    \( \langle [t = A \land c = 0] \cdot 1 + [t = A \land c = 1] \cdot \frac{a}{a+b-ab} + [t = B \land c = 1] \cdot \frac{(1-b)a}{a+b-ab} \rangle \)
    while (c = 1) {
        \( \langle [t = A \land c = 1] \cdot \frac{a}{a+b-ab} + [t = B \land c = 1] \cdot \frac{(1-b)a}{a+b-ab} \rangle \)
        \( \langle [t = A \land c \neq 1] \cdot a + [t = A \land c = 1] \cdot \frac{a}{a+b-ab} + [t = B \land c = 0] \cdot (1-b) + [t = B \land c = 1] \cdot \frac{(1-b)a}{a+b-ab} \rangle \)
        if (t = A) {
            (c := 0 [a] t := B);
        } else {
            (c := 0 [b] t := A);
        }
    \( \langle [t = A \land c = 0] \cdot 1 + [t = A \land c = 1] \cdot \frac{a}{a+b-ab} + [t = B \land c = 1] \cdot \frac{(1-b)a}{a+b-ab} \rangle \)
    }
    \( \langle [c \neq 1] \cdot \left( [t = A \land c = 0] \cdot 1 + [t = A \land c = 1] \cdot \frac{a}{a+b-ab} + [t = B \land c = 1] \cdot \frac{(1-b)a}{a+b-ab} \right) \rangle \)
    \( \langle [t = A] \rangle \)
    return t; // the survivor
}
```
When one starts nondeterministically

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

```c
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;
}
```

```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;
            (skip [q] f := 1);
        }
    }
    return x;
}
```

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

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