Analyzing Probabilistic Programs:
Pushing the Limits of Automation

Joost-Pieter Katoen

Software Modeling and Verification Group
RWTH Aachen University

joint work with Friedrich Gretz and Annabelle McIver

February 6, 2012

RWTH AACHEN UNIVERSITY

Overview

Once upon a time .....

Duelling cowboys

```
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?
}
```

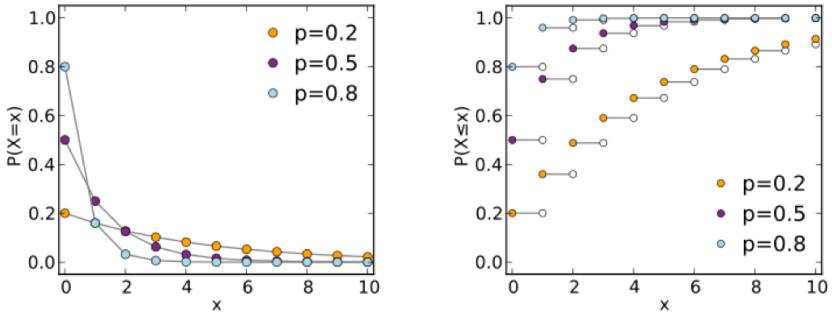
Claim:

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 $0 < p \leqslant 1$. 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 $X - Y$

```
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 (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?

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.

Duelling cowboys

```
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?
}
```

We can infer:

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

Program equivalence

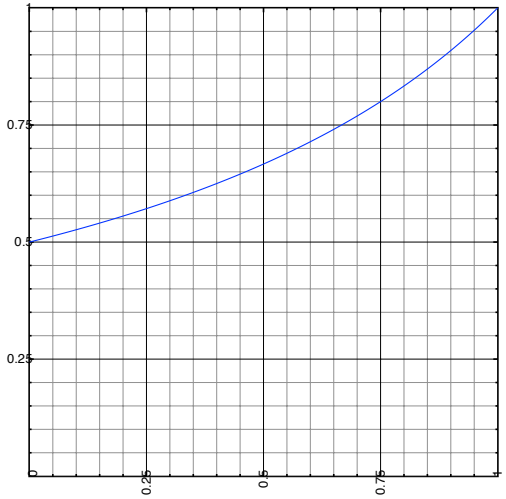
```
int XminY1(float p, q){
  int  x, f := 0, 0;
  while (f = 0) {
    (x +:= 1 [p] f := 1);
  }
  f := 0;
  while (f = 0) {
    (x -:= 1 [q] f := 1);
  }
  return x;
}
```

```
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 $q = \frac{1}{2-p}$.

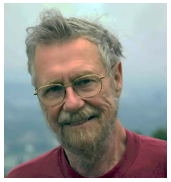
Graphically this means ...



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

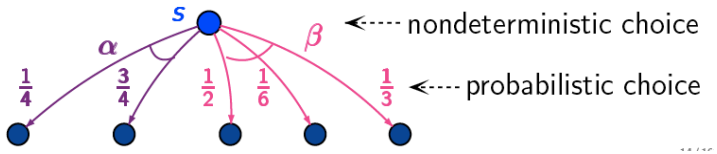
Overview

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 $S_0 \subseteq S$, $S_0 \neq \varnothing$
- $\longrightarrow \subseteq S \times \mathit{Dist}(S)$ is a transition relation

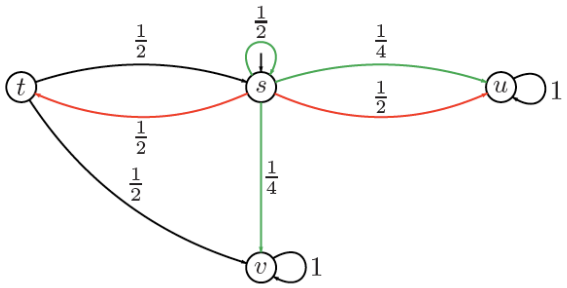


Notation:

$s \to \mu$ denotes $(s, \mu) \in \longrightarrow$ and $s \to t$ denotes $s \to \mu$ with $\mu(t) = 1$. Let $\mathit{Dist}(s) = \{\, \mu \mid s \to \mu \,\}$. 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 $\mathit{Dist}(s) \neq \varnothing$, nondeterministically select $\mu \in \mathit{Dist}(s)$
3. The next state t is randomly chosen with probability $\mu(t)$.
4. If $\mathit{Dist}(t) = \varnothing$, 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.[1]

Policy

Function $\mathfrak{P} : S \to \mathit{Dist}(S)$ is a positional policy for MDP $\mathcal{M} = (S, S_0, \longrightarrow)$ with $\mathfrak{P}(s) \in \mathit{Dist}(s)$ for all $s \in S$.

Alternating sequence

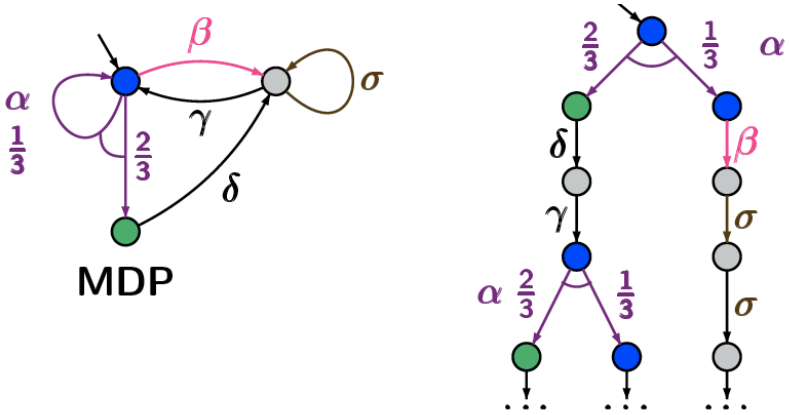
$$\pi \;=\; s_0 \xrightarrow{\mu_0} s_1 \xrightarrow{\mu_1} \ldots$$

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

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

[1] Also 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 $P \in \text{pGCL}$ 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 $Q \in \text{pGCL}$ or $Q = \text{exit}$
- Initial states are tuples $\langle P, \eta \rangle$ where η fulfils the initial conditions
- Transition relation $\to$ is the smallest relation satisfying the inference rules (cf. next slide)

Structured operational semantics

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

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

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

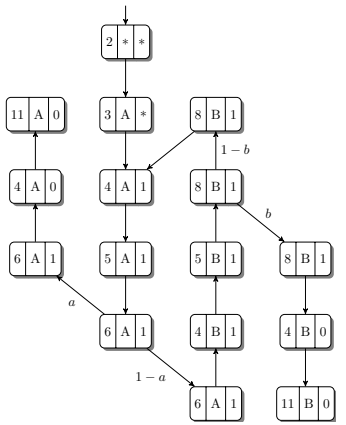
$$\langle P[p]Q, \eta \rangle \to \mu \text{ with } \mu(\langle P, \eta \rangle) = p \text{ and } \mu(\langle Q, \eta \rangle) = 1-p$$

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

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

MDP of duelling cowboys

```
int cowboyDuel(float a, b) {
  int t := A;
  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 $E \Rightarrow wp(P, F)$.

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

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

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

Expectations

Weakest pre-expectation

An expectation is a 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 $e \ \leqslant \ wp(P, f)$, where $\leqslant$ 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.

Expectation transformer semantics of pGCL

Syntax

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

Semantics $wp(P, f)$

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

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

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

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

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

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 $r : S \to \mathbb{Z}$ 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 $\pi = s_0 \xrightarrow{\mu_0} s_1 \xrightarrow{\mu_1} \ldots$ 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 cost 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 $r_T(\pi) = 0$.

Cost-bounded reachability

Expected reward for reachability

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

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

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

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

$$CExpRew(s \models \Diamond T) = \min_{\mathfrak{P}} \sum_{c=-\infty}^{\infty} c \cdot \frac{Pr^{\mathfrak{P}}\{\, \pi \in Paths^{\mathfrak{P}}(s, \Diamond T) \mid r_T(\pi) = c \,\}}{Pr^{\mathfrak{P}}(s \models \Diamond T)}$$

where $\mathfrak{P}$ is a policy such that $Pr^{\mathfrak{P}}(s \models \Diamond T) > 0$.

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 η, and post-expectation f:

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

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

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

Thus, $wp(P, f)$ evaluated at η 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.

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 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:

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

   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.

Play the game

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

```
void playGame {
  flip;
  while ¬(all(♡) ∨ all(♢)) {
     flip;
  }
}
```

Example loop invariant

Let $Q' = 1 \cdot [all(\heartsuit)] + \frac{1}{2} \cdot [all(\diamondsuit)]$

- Invariant $I = \frac{3}{4} \cdot [\neg all(\heartsuit) \wedge \neg all(\diamondsuit)] + 1 \cdot [all(\heartsuit)] + \frac{1}{2} \cdot [all(\diamondsuit)]$.
- As $Q' = [all(\heartsuit) \vee all(\diamondsuit)] \cdot I$ we have $\{\, I \,\}$ *loop* $\{\, Q' \,\}$
- It follows $wlp(init, I) = \frac{3}{4}$.
- 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 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.

Disjoint normal form

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

Disjoint normal form

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

$$\bigwedge_{0<n\leqslant N} P_{in} \;\wedge\; \bigwedge_{0<n\leqslant N} P_{jn} \;=\; \mathit{false}$$

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:

$$Q_{MN} \;\leqslant\; Q'_{KL}$$

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

$$P_m \;\wedge\; P'_k \;\Rightarrow\; (Q_m - Q'_k \;\leqslant\; 0)$$

$$P_m \;\wedge\; \left(\bigwedge_{k\in[1..K]} \neg P'_k\right) \;\Rightarrow\; Q_m \;\leqslant\; 0$$

Any inequality $Q \leqslant Q'$ between propositionally linear expressions is equivalent to a finite Boolean expression over linear constraints.

Obtaining existentially quantified FO-formulas

Motzkin's transposition theorem is one of the deepest results in the part of mathematics dealing with linear inequalities [Nemirovski & Roos, Encyclopedia of Optimization, 2009]

Motzkin's transposition theorem (1936)

Let A, A' be matrices, b, b' column vectors, and x a column vector of variables.

If $A{\cdot}x \leqslant b$ and $A'{\cdot}x < b'$ is unsatisfiable, then there exist row vectors λ, λ' with:

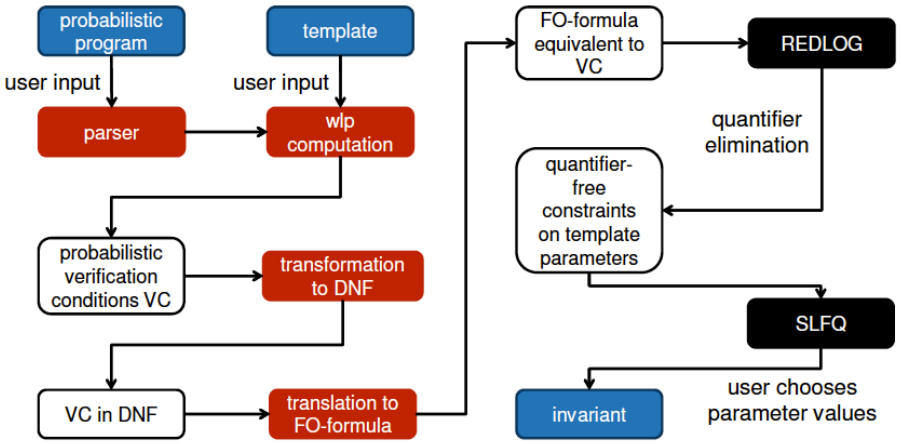
$$\lambda \;\geqslant\; 0 \quad \text{and} \quad \lambda' \geqslant 0 \quad \text{and} \quad \lambda{\cdot}A + \lambda'{\cdot}A' = 0$$

and either

1. $\lambda{\cdot}b + \lambda'{\cdot}b' > 0$, or
2. some entry of λ' is strictly positive and $\lambda{\cdot}b + \lambda'{\cdot}b' \geqslant 0$.

(λ and λ' form a witness of $A{\cdot}x \geqslant b$ and $A'{\cdot}x > b'$ being unsatisfiable.)

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

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

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

$\mathcal{T} = [\mathrm{t} = A \wedge \mathrm{c} = 0] \cdot 1 + [\mathrm{t} = A \wedge \mathrm{c} = 1] \cdot \dfrac{a}{a + b - a \cdot b} + [\mathrm{t} = B \wedge \mathrm{c} = 1] \cdot \dfrac{(1-b)a}{a + b - a \cdot b}$

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

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

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

	(combined) bisimilarity	(combined) similarity
PDA vs. finite TS	PSPACE-complete	EXPTIME-complete
pPDA vs. finite pTS	EXPTIME-complete	EXPTIME-complete

Overview

Epilogue

Take-home message

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

$\Rightarrow$ 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.

Fin.

Congratulations