Predictable Probabilistic Programming
by Deductive Verification

Joost-Pieter Katoen

Workshop on Software Correctness and Reliability, ETH Zurich, 2017
Overview

1. Introduction
2. Probabilistic guarded command language
3. Predicate transformers
4. Termination
5. Synthesizing loop invariants
6. Recursion
7. Epilogue
Perspective

“There are several reasons why probabilistic programming could prove to be revolutionary for machine intelligence and scientific modelling.”

---

1 Zoubin Ghahramani leads the Cambridge Machine Learning Group, and holds positions at CMU, UCL, and the Alan Turing Institute.
Probabilistic programs

What?

They are programs with random assignments and conditioning

Why?

- Random assignments: to describe randomised algorithms
- Conditioning: to describe stochastic decision making
Sorting by flipping coins
Sorting by flipping coins

QuickSort:

\[
\begin{align*}
\text{QS}(A) &= \ \\
\text{if } |A| \leq 1 \{ \text{return } A; \} \\
i &= \text{ceil}(|A|/2); \\
A_< &= \{a \in A \mid a < A[i]\}; \\
A_> &= \{a \in A \mid a > A[i]\}; \\
\text{return } \text{QS}(A_<) + A[i] + \text{QS}(A_>)
\end{align*}
\]

Worst case complexity: \(O(N^2)\) comparisons
Sorting by flipping coins

QuickSort:

\[
\text{QS}(A) = \begin{cases} 
\text{return } A & \text{if } |A| \leq 1 \\
\text{ceil}(|A|/2); \\
A_\text{<} := \{ a \in A \mid a < A[1] \}; \\
A_\text{>} := \{ a \in A \mid a > A[1] \}; \\
\text{return } \text{QS}(A_\text{<}) + A[i] + \text{QS}(A_\text{>}) 
\end{cases}
\]

Worst case complexity: \( O(N^2) \) comparisons

Randomised QuickSort:

\[
\text{rQS}(A) = \begin{cases} 
\text{return } A & \text{if } |A| \leq 1 \\
i := \text{Unif}[1...|A|]; \\
A_\text{<} := \{ a \in A \mid a < A[i] \}; \\
A_\text{>} := \{ a \in A \mid a > A[i] \}; \\
\text{return } \text{rQS}(A_\text{<}) + A[i] + \text{rQS}(A_\text{>}) 
\end{cases}
\]

Worst case complexity: \( O(N \log N) \) expected comparisons
Probabilistic graphical models
Student’s mood after an exam

How likely does a well-prepared student end up with a bad mood?
Ecology

When to purchase irrigation rights or impose pumping restrictions?
Rethinking the Bayesian approach

“In particular, the graphical model formalism that ushered in an era of rapid progress in AI has proven inadequate in the face of [these] new challenges.

[Daniel Roy, 2011]
Rethinking the Bayesian approach

“In particular, the graphical model formalism that ushered in an era of rapid progress in AI has proven inadequate in the face of [these] new challenges.

A promising new approach that aims to bridge this gap is probabilistic programming, which marries probability theory, statistics and programming languages”

MIT/EECS George M. Sprowls Doctoral Dissertation Award
Applications
Languages

Languages:

Probabilistic C
ProbLog
Church
webPPL
Figaro
PyMC
Tabular
R2

probabilistic-programming.org

A. Pfeffer

N. Goodman
Roadmap

1. Introduction
2. Probabilistic guarded command language
3. Predicate transformers
4. Termination
5. Synthesizing loop invariants
6. Recursion
7. Epilogue
Overview

1. Introduction

2. Probabilistic guarded command language

3. Predicate transformers

4. Termination

5. Synthesizing loop invariants

6. Recursion

7. Epilogue
Probabilistic GCL

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

empty statement
abortion
assignment
conditioning
sequential composition
choice
probabilistic choice
iteration
Let’s start simple

\[
\begin{align*}
x & := 0 \ [0.5] \ x := 1; \\
y & := -1 \ [0.5] \ y := 0
\end{align*}
\]

This program admits four runs and yields the outcome:

\[
Pr[x=0, y=0] = Pr[x=0, y=-1] = Pr[x=1, y=0] = Pr[x=1, y=-1] = \frac{1}{4}
\]
A loopy program

For $0 < p < 1$ an arbitrary probability:

```cpp
bool c := true;
int i := 0;
while (c) {
    i++;
    (c := false [p] c := true)
}
```

The loopy program models a geometric distribution with parameter $p$.

$$Pr[i = N] = (1-p)^{N-1} \cdot p \quad \text{for } N > 0$$
On termination

```c
bool c := true;
int i := 0;
while (c) {
    i++;
    (c := false [p] c := true)
}
```

This program does not always terminate. It almost surely terminates.
Conditioning

\[ P(A|B) = \frac{P(B|A)P(A)}{P(B)} \]
Let’s start simple

\[\begin{align*}
x &:= 0 \quad [0.5] \quad x := 1; \\
y &:= -1 \quad [0.5] \quad y := 0; \\
\text{observe} \quad (x+y = 0)
\end{align*}\]
Let's start simple

\[
x := 0 \ [0.5] \ x := 1;
y := -1 \ [0.5] \ y := 0;
\text{observe} \ (x+y = 0)
\]

This program blocks two runs as they violate \(x+y = 0\). Outcome:

\[
Pr[x=0, y=0] = Pr[x=1, y=-1] = \frac{1}{2}
\]
Let’s start simple

```plaintext
x := 0 [0.5] x := 1;
y := -1 [0.5] y := 0;
observe (x+y = 0)
```

This program blocks two runs as they violate \( x+y = 0 \). Outcome:

\[
Pr[x=0, y=0] = Pr[x=1, y=-1] = \frac{1}{2}
\]

Observations thus normalize the probability of the “feasible” program runs.
A loopy program

For $0 < p < 1$ an arbitrary probability:

```cpp
bool c := true;
int i := 0;
while (c) {
    i++;
    (c := false [p] c := true)
}
observe (odd(i))
```
A loopy program

For $0 < p < 1$ an arbitrary probability:

```c
bool c := true;
int i := 0;
while (c) {
    i++;
    (c := false [p] c := true)
}
```

`observe (odd(i))`

The feasible program runs have a probability

$$
\sum_{N \geq 0} (1-p)^{2N} \cdot p = \frac{1}{2 - p}
$$
A loopy program

For $0 < p < 1$ an arbitrary probability:

```plaintext
bool c := true;
int i := 0;
while (c) {
    i++;
    (c := false [p] c := true)
}
observe (odd(i))
```

The feasible program runs have a probability

$$\sum_{N\geq0} (1-p)^{2N} \cdot p = \frac{1}{2-p}$$

This program models the distribution:

$$Pr[i = 2N+1] = (1-p)^{2N} \cdot p \cdot (2-p) \quad \text{for } N \geq 0$$

$$Pr[i = 2N] = 0$$
Overview

1. Introduction
2. Probabilistic guarded command language
3. Predicate transformers
4. Termination
5. Synthesizing loop invariants
6. Recursion
7. Epilogue
The need for **predictable** probabilistic programming

- Lifting simulation techniques for graphical models to programs is difficult
  - Monte Carlo Markov Chain simulation of R2 and STAN is erroneous
The need for **predictable** probabilistic programming

- Lifting simulation techniques for graphical models to programs is difficult
  - Monte Carlo Markov Chain simulation of R2 and STAN is erroneous

- Simulative analyses come with **weak guarantees**
The need for **predictable** probabilistic programming

- **Lifting simulation** techniques for graphical models to programs is difficult
  - Monte Carlo Markov Chain simulation of R2 and STAN is erroneous

- Simulative analyses come with **weak guarantees**

- The analyses are not guaranteed to **terminate**
  - Usual assumptions about termination are much too strong
The need for **predictable** probabilistic programming

- Lifting simulation techniques for graphical models to programs is difficult
  - Monte Carlo Markov Chain simulation of R2 and STAN is erroneous

- Simulative analyses come with weak guarantees

- The analyses are not guaranteed to terminate
  - Usual assumptions about termination are much too strong

Verifiable programs are preferable to simulative guarantees.
Weakest preconditions

**Weakest precondition**

An **predicate** maps program states onto Booleans.
# Weakest preconditions

**Weakest precondition**

[Dijkstra 1975]

An **predicate** maps program states onto Booleans.

A **predicate transformer** is a total function between two predicates.
Weakest preconditions

Weakest precondition

An **predicate** maps program states onto Booleans.

A **predicate transformer** is a total function between two predicates.

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 $F$."

[Dijkstra 1975]
Weakest preconditions

<table>
<thead>
<tr>
<th>Weakest precondition</th>
<th>[Dijkstra 1975]</th>
</tr>
</thead>
<tbody>
<tr>
<td>An predicate maps program states onto Booleans.</td>
<td></td>
</tr>
<tr>
<td>A predicate transformer is a total function between two predicates.</td>
<td></td>
</tr>
<tr>
<td>The predicate transformer $wp(P, F)$ for program $P$ and postcondition $F$ yields the &quot;weakest&quot; precondition $E$ on the initial state of $P$ ensuring that the execution of $P$ terminates in a final state satisfying $F$.</td>
<td></td>
</tr>
<tr>
<td>Hoare triple ${E} P {F}$ holds for total correctness iff $E \Rightarrow wp(P, F)$.</td>
<td></td>
</tr>
</tbody>
</table>
Weakest preconditions

An predicate maps program states onto Booleans.

A predicate transformer is a total function between two predicates.

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 pre-condition $wlp(P, F) = "wp(P, F) or P diverges"$. 
Predicate transformer semantics of Dijkstra’s GCL

### Syntax

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

\[ wp(P, F) = \begin{cases} F & \text{if } F \text{ is false} \\ F[x := E] & \text{if } F \text{ is true} \\ wp(P_1, wp(P_2, F)) & \text{if } \neg G \implies wp(P_1, F) \lor G \implies wp(P_2, F) \\ wp(P_1, F) \land wp(P_2, F) & \text{if } (G \land wp(P, X)) \lor (\neg G \land F) \\ \end{cases} \]

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

wlp-semantics differs from wp-semantics only for `while` and `abort`.
# 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`
- $F[x := E]$
- $wp(P_1, wp(P_2, F))$
- $(G \land wp(P_1, F)) \lor (\neg G \land wp(P_2, F))$
- $wp(P_1, F) \land wp(P_2, F)$
- $\mu X. ((G \land wp(P, X)) \lor (\neg G \land F))$
Predicate transformer semantics of Dijkstra’s GCL

### Syntax
- skip
- abort
- \( x := E \)
- \( P_1 ; P_2 \)
- if \((G)P_1\) else \(P_2\)
- \( P_1 \ [\] \ P_2 \)
- while \((G)P\)

### Semantics \( wp(P, F) \)
- \( F \)
- false
- \( F[x := E] \)
- \( wp(P_1, wp(P_2, F)) \)
- \((G \land wp(P_1, F)) \lor \lnot G \land wp(P_2, F)) \)
- \( wp(P_1, F) \land wp(P_2, F) \)
- \( \mu X. ((G \land wp(P, X)) \lor \lnot G \land F) \)

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

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

Weakest pre-expectation

An expectation\(^2\) maps program states onto non-negative reals (extended with \(\infty\)). It is the quantitative analogue of a predicate.

\[^2\neq\text{expectations in probability theory.}\]
**Expectations**

**Weakest pre-expectation** [McIver & Morgan 2004]

An expectation\(^2\) maps program states onto non-negative reals (extended with ∞). It is the quantitative analogue of a predicate. Let \( f \leq g \) iff for every state \( s \) it holds \( f(s) \leq g(s) \).

\(^2\) ≠ expectations in probability theory.
Expectations

**Weakest pre-expectation** [McIver & Morgan 2004]

An expectation \(^2\) maps program states onto non-negative reals (extended with \(\infty\)). It is the quantitative analogue of a predicate. Let \(f \leq g\) iff for every state \(s\) it holds \(f(s) \leq g(s)\).

An expectation transformer is a total function between two expectations.

\(^2\) ≠ expectations in probability theory.
## Expectations

<table>
<thead>
<tr>
<th>Weakest pre-expectation</th>
<th>[McIver &amp; Morgan 2004]</th>
</tr>
</thead>
<tbody>
<tr>
<td>An expectation(^2) maps program states onto non-negative reals (extended with (\infty)). It is the quantitative analogue of a predicate. Let (f \leq g) iff for every state (s) it holds (f(s) \leq g(s)).</td>
<td></td>
</tr>
<tr>
<td>An expectation transformer is a total function between two expectations.</td>
<td></td>
</tr>
<tr>
<td>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).</td>
<td></td>
</tr>
</tbody>
</table>

\(^2\) ≠ expectations in probability theory.
Expectations

**Weakest pre-expectation**

An expectation\(^2\) maps program states onto non-negative reals (extended with \(\infty\)). It is the quantitative analogue of a predicate. Let \(f \leq g\) iff for every state \(s\) it holds \(f(s) \leq g(s)\).

An expectation transformer is a total function between two expectations. 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 \(e \leq wp(P, f)\).

\(^2\) ≠ expectations in probability theory.
## Expectations

### Weakest pre-expectation

<table>
<thead>
<tr>
<th><strong>Weakest pre-expectation</strong></th>
<th><strong>[McIver &amp; Morgan 2004]</strong></th>
</tr>
</thead>
</table>

An expectation\(^2\) maps program states onto non-negative reals (extended with \(\infty\)). It is the quantitative analogue of a predicate. Let \(f \leq g\) iff for every state \(s\) it holds \(f(s) \leq g(s)\).

An **expectation transformer** is a total function between two expectations.

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 \(e \leq wp(P, f)\).

Weakest liberal pre-expectation \(wlp(P, f) = "wp(P, f) + Pr[P \text{ diverges}]"\).

\(^2\) ≠ expectations in probability theory.
Expectation transformer semantics of \( pGCL \)

**Syntax**

- `skip`
- `abort`
- `x := E`
- `observe (G)`
- `P1 ; P2`
- `if (G)P1 else P2`
- `P1 [p] P2`
- `while (G)P`
# Expectation transformer semantics of pGCL

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Semantics $\text{wp}(P, f)$</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>▶ observe $(G)$</td>
<td>▶ $[G] \cdot f$</td>
</tr>
<tr>
<td>▶ $P_1 ; P_2$</td>
<td>▶ $\text{wp}(P_1, \text{wp}(P_2, f))$</td>
</tr>
<tr>
<td>▶ if $(G)P_1$ else $P_2$</td>
<td>▶ $[G] \cdot \text{wp}(P_1, f) + [\neg G] \cdot \text{wp}(P_2, f)$</td>
</tr>
<tr>
<td>▶ $P_1 [p] P_2$</td>
<td>▶ $p \cdot \text{wp}(P_1, f) + (1 - p) \cdot \text{wp}(P_2, f)$</td>
</tr>
<tr>
<td>▶ while $(G)P$</td>
<td>▶ $\mu X. ([G] \cdot \text{wp}(P, X) + [\neg G] \cdot f)$</td>
</tr>
</tbody>
</table>

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

$\text{wp}$-semantics differs from $\text{wp}$-semantics only for `while` and `abort`.
### Expectation transformer transformer semantics of pGCL

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

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

\( \mu \) is the least fixed point operator wrt. the ordering \( \leq \) on expectations.
# Expectation transformer semantics of pGCL

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

## Semantics \(wp(P, f)\)
- `f`
- `0`
- `f[x := E]`
- `[G] \cdot f`
- `wp(P_1, wp(P_2, f))`
- `[G] \cdot wp(P_1, f) + [\neg G] \cdot 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**  
Predictable Probabilistic Programming  
26/64
Wp = conditional rewards in Markov chains

For program $P$, input $s$ and expectation $f$:

$$\frac{wp(P, f)(s)}{wlp(P, 1)(s)} = \text{ER}^{[P]}(s, \Diamond(sink) \cap \neg \Diamond(\#))$$

The ratio of $wp(P, f)$ over $wlp(P, 1)$ for input $s$ equals\(^3\) the conditional expected reward to reach a successful terminal state $\langle sink \rangle$ while satisfying all observations in $P$'s MC when starting with $s$.

\(^3\) Either both sides are equal or both sides are undefined.
\[ W_p = \text{conditional rewards in Markov chains} \]

For program \( P \), input \( s \) and expectation \( f \):

\[
\frac{w_p(P, f)(s)}{w_l(P, 1)(s)} = \text{ER}_{[P]}(s, \Diamond\langle sink \rangle \cap \neg \Diamond \langle \ddagger \rangle)
\]

The ratio of \( w_p(P, f) \) over \( w_l(P, 1) \) for input \( s \) equals\(^3\) the conditional expected reward to reach a successful terminal state \( \langle sink \rangle \) while satisfying all observations in \( P \)'s MC when starting with \( s \).

Conditional expected rewards in finite MCs can be computed in polynomial time.

---

\(^3\) Either both sides are equal or both sides are undefined.
Student’s mood after an exam

How likely does a well-prepared student end up with a bad mood?
Programs for Bayesian networks

- Take a topological sort of the BN’s vertices, e.g., $D; P; G; M$

- Map each conditional probability table (aka: node) to a program, e.g.:

```java
if (xD = 0 && xP = 0) {
    xG := 0 [0.95] xG := 1
} else if (xD = 1 && xP = 1) {
    xG := 0 [0.05] xG := 1
} else if (xD = 0 && xP = 1) {
    xG := 0 [0.5] xG := 1
} else if (xD = 1 && xP = 0) {
    xG := 0 [0.6] xG := 1
}
```

- Condition on the evidence, e.g., for $P = 1$ we get:

```java
repeat { progD ; progP; progG ; progM } until (P=1)
```
Soundness

Correctness of BN programs

For BN $B$ over variables $V$ with evidence $obs$ over $O \subseteq V$ and value $v$ for node (and input) $v$:

$$wp(prog(B, obs), \bigwedge_{v \in V \setminus O} x_v = v) = Pr\left(\bigwedge_{v \in V \setminus O} v = v \mid \bigwedge_{o \in O} o = o\right)$$

where $prog(B, obs)$ equals `repeat progB until (\bigwedge_{o \in O} x_o = obs(o))`. 
## Soundness

### Correctness of BN programs

For BN $B$ over variables $V$ with evidence $obs$ over $O \subseteq V$ and value $v$ for node (and input) $v$:

$$wp(prog(B, obs), \bigwedge_{v \in V \setminus O} x_v = v) = Pr\left(\bigwedge_{v \in V \setminus O} v = v \mid \bigwedge_{o \in O} o = o\right)$$

where $prog(B, obs)$ equals `repeat progB until (\bigwedge_{o \in O} x_o = obs(o))`.

Ergo: exact Bayesian inference by wp-reasoning!, e.g.,
Soundness

Correctness of BN programs

For BN $B$ over variables $V$ with evidence $obs$ over $O \subseteq V$ and value $v$ for node (and input) $v$:

$$wp(prog(B, obs), \bigwedge_{v \in V \setminus O} x_v = v) = Pr\left(\bigwedge_{v \in V \setminus O} v = v \mid \bigwedge_{o \in O} o = o\right)$$

where $prog(B, obs)$ equals $repeat \ progB \ until \ (\bigwedge_{o \in O} x_o = obs(o))$.

Ergo: exact Bayesian inference by $wp$-reasoning!, e.g.,

$$wp(P_{mood}, [x_D = 0 \land x_G = 0 \land x_M = 0]) = \frac{Pr(D = 0, G = 0, M = 0, P = 1)}{Pr(P = 1)} \approx 0.27$$
Soundness

Correctness of BN programs

For BN $B$ over variables $V$ with evidence $obs$ over $O \subseteq V$ and value $v$ for node (and input) $v$:

$$wp\left(\text{prog}(B, obs), \bigwedge_{v \in V \setminus O} x_v = v\right) = Pr\left(\bigwedge_{v \in V \setminus O} v = v \mid \bigwedge_{o \in O} o = o\right)$$

where $\text{prog}(B, obs)$ equals $\text{repeat } \text{progB until } (\bigwedge_{o \in O} x_o = obs(o))$.

Ergo: exact Bayesian inference by wp-reasoning!, e.g.,

$$wp(P_{mood}, [x_D = 0 \land x_G = 0 \land x_M = 0]) = \frac{Pr(D = 0, G = 0, M = 0, P = 1)}{Pr(P = 1)} \approx 0.27$$

As loops in BN programs are independent, precise wp’s can be provided
Overview

1. Introduction
2. Probabilistic guarded command language
3. Predicate transformers
4. Termination
5. Synthesizing loop invariants
6. Recursion
7. Epilogue
On termination

```c
bool c := true;
int i := 0;
while (c) {
    i ++;
    (c := false [p] c := true)
}
```

This program does not always terminate. It almost surely terminates.
Nuances of termination

...... certain termination
Nuances of termination

...... certain termination

...... termination with probability one

$$\Rightarrow$$ almost-sure termination
Nuances of termination

...... certain termination

...... termination with probability one
     ⇒ almost-sure termination

...... in an expected finite number of steps
     ⇒ positive almost-sure termination
Positive almost-sure termination

For $0 < p < 1$ an arbitrary probability:

```c
bool c := true;
int i := 0;
while (c) {
    i++;
    (c := false [p] c := true)
}
```

This program almost surely terminates. In finite expected time. Despite the possibility of divergence.
**Negative almost-sure termination**

Consider the one-dimensional symmetric random walk:

```java
int x := 10;
while (x > 0) {
    (x := x-1 [0.5] x := x+1)
}
```

This program *almost surely* terminates but requires an *infinite* expected time to do so.
Compositionality

Consider the two probabilistic programs:
Compositionality

Consider the two probabilistic programs:

```plaintext
int x := 1;
bool c := true;
while (c) {
    c := false [0.5] c := true;
    x := 2*x
}
```

Finite expected termination time
Consider the two probabilistic programs:

```c
int x := 1;
bool c := true;
while (c) {
    c := false [0.5] c := true;
    x := 2*x
}
```

Finite expected termination time

```c
while (x > 0) {
    x := x-1
}
```

Finite termination time
**Compositionality**

Consider the two probabilistic programs:

```plaintext
int x := 1;
bool c := true;
while (c) {
    c := false [0.5] c := true;
    x := 2*x
}

while (x > 0) {
    x := x-1
}
```

*Finite expected termination time*

Running these programs in sequence yields an *infinite* expected termination time.
Termination is hard

Almost-sure termination is “more undecidable” than certain termination
Termination is hard

Almost-sure termination is “more undecidable” than certain termination

Almost-sure termination is “as hard” as the universal halting problem
Termination is hard

Almost-sure termination is “more undecidable” than certain termination

Almost-sure termination is “as hard” as the universal halting problem

Positive almost-sure termination is “one degree more undecidable” than almost-sure termination
Termination proofs: the classical case

$V(s^i)$

→ loop iterations
Termination proofs: the classical case

\[ V(s^i) \]

Arrival at 0 guaranteed by well-foundedness of \( \succ \).
Termination proofs: the classical case

$$V(s^i)$$

arrival at 0 guaranteed by well-foundedness of >
Proving almost-sure termination

The symmetric random-walk is troublesome for many proof rules:

```plaintext
while (x > 0) {
    (x := x-1 [0.5] x := x+1)
}
```
Proving almost-sure termination

The symmetric random-walk is troublesome for many proof rules:

```
while (x > 0) {
    (x := x-1 [0.5] x := x+1)
}
```

A loop iteration decreases $x$ by one with probability $1/2$
Proving almost-sure termination

The symmetric random-walk is troublesome for many proof rules:

```plaintext
while (x > 0) {
    (x := x-1 [0.5] x := x+1)
}
```

A loop iteration decreases $x$ by one with probability $\frac{1}{2}$

This observation is enough to witness almost-sure termination!
Proving almost-sure termination

Goal: prove a.s.–termination of \( \text{while}(G) \ P \), for probabilistic program \( P \)
Proving almost-sure termination

Goal: prove a.s.–termination of while(G) P, for probabilistic program P

Ingredients:

- A supermartingale V mapping states onto non-negative reals
  - Executing P on a state s satisfying G does not increase V’s expected value
  - Loop iteration ceases if V(s) = 0

Then:
program while(G) P terminates almost surely on every input
Proving almost-sure termination

**Goal:** prove a.s.–termination of $\texttt{while}(G) \ P$, for probabilistic program $P$

**Ingredients:**
- A *supermartingale* $V$ mapping states onto non-negative reals
  - Executing $P$ on a state $s$ satisfying $G$
    does not increase $V$’s expected value
- Loop iteration ceases if $V(s) = 0$
- and a *progress* condition: on each loop iteration
  - $V$’s value $v$ decreases by $\geq d(v)$ with probability $\geq p(v)$
  - with antitone function $p$ (“probability”) on $V$’s values
  - and antitone function $d$ (“decrease”) on $V$’s values
Proving almost-sure termination

**Goal:** prove a.s.–termination of while(G) P, for probabilistic program P

**Ingredients:**
- A supermartingale $V$ mapping states onto non-negative reals
  - Executing P on a state $s$ satisfying G does not increase $V$’s expected value
  - Loop iteration ceases if $V(s) = 0$
- and a **progress** condition: on each loop iteration
  - $V$’s value $v$ decreases by $\geq d(v)$ with probability $\geq p(v)$
    - with antitone function $p$ (“probability”) on $V$’s values
    - and antitone function $d$ (“decrease”) on $V$’s values

**Then:** program while(G) P terminates almost surely on every input
Proving almost-sure termination

\[ V(s^i) \]

\[ \rightarrow \text{loop iterations} \]
Proving almost-sure termination

$$V(s^i)$$

The closer to termination, the more $$V$$ decreases and this becomes more likely. A.s. arrival at 0 guaranteed by our proof rule.
Proving almost-sure termination

$V(s^i) \rightarrow \text{loop iterations}$

$V(s^1) \rightarrow d(V(s^1))$

with prob. $\geq p(V(s^1))$

The closer to termination, the more $V$ decreases and this becomes more likely.
Proving almost-sure termination

\[ V(s^i) \]

with prob. \( \geq p(V(s^1)) \)

\[ V(s^1) \]

\[ V(s^2) \]

\[ V(s^4) \]

\[ V(s^5) \]

\[ s^0, s^1, s^2, s^3, s^4, s^5, s^6, s^7, s^8, s^9 \]

→ loop iterations

The closer to termination, the more \( V \) decreases and this becomes more likely
Proving almost-sure termination

\[ V(s^i) \]

- \( V(s^1) \) with prob. \( \geq p(V(s^1)) \)
- \( V(s^2) \) with prob. \( \geq p(V(s^2)) \)
- \( V(s^4) \) with prob. \( \geq p(V(s^4)) \)
- \( V(s^5) \)

\( s^0 \to s^1 \to s^2 \to s^3 \to s^4 \to s^5 \to s^6 \to s^7 \to s^8 \to s^9 \)

\rightarrow \text{loop iterations}

\text{with prob.} \geq p(V(s^1))

\text{by antitone}

\text{a.s. arrival at 0 guaranteed by our proof rule}

The closer to termination, the more \( V \) decreases and this becomes more likely
Proving almost-sure termination

\[ \text{with prob. } \geq p(V(s^1)) \]

\[ \text{with prob. } \geq p(V(s^4)) \]

\[ d(V(s^1)) \]

\[ d(V(s^4)) \]

\[ d(V1) \leq d(V4) \]

by antitone \( d \)
Proving almost-sure termination

\[ V(s^i) \]

with prob. \( \geq p(V(s^1)) \)

\[ d(V(s^1)) \]

\[ V(s^2) \]

with prob. \( \geq p(V(s^4)) \)

\[ d(V(s^4)) \]

\[ V(s^4) \]

\[ d(V^1) \leq d(V^4) \]

by antitone \( d \)

\[ V(s^5) \]

\[ \cdots \]

\[ V(s^9) \]

\[ \cdots \]

\[ \cdots \]

\[ \cdots \]

by antitone \( p \)

\[ \text{a.s. arrival at 0 guaranteed} \]

The closer to termination, the more \( V \) decreases and this becomes more likely.
Proving almost-sure termination

\[ p(V_1) \leq p(V_4) \]

by antitone \( p \)

\[ d(V_1) \leq d(V_4) \]

by antitone \( d \)

The closer to termination, the more \( V \) decreases and this becomes more likely.

\[ a.s. \text{ arrival at 0 guaranteed by our proof rule} \]
Proving almost-sure termination

\[ V(s^i) \]

\[ V(s^1) \rightarrow d(V(s^1)) \]
\[ V(s^4) \rightarrow d(V(s^4)) \]

The closer to termination, the more \( V \) decreases and this becomes more likely.

\[ p(V1) \leq p(V4) \]
by antitone \( p \)

\[ d(V1) \leq d(V4) \]
by antitone \( d \)
The symmetric random walk

Recall the program:

\[
\text{while } (x > 0) \{ x := x-1 \ [0.5] x := x+1 \}
\]
The symmetric random walk

Recall the program:

\[
\textbf{while } (x > 0) \{ x := x-1 \ [0.5] \ x := x+1 \}
\]

Witnesses of almost-sure termination:

- \( V = x \)
- \( p(v) = \frac{1}{2} \) and \( d(v) = 1 \)
The symmetric random walk

- Recall the program:

  ```
  while (x > 0) { x := x-1 [0.5] x := x+1 }
  ```

- Witnesses of almost-sure termination:
  - \( V = x \)
  - \( p(v) = \frac{1}{2} \) and \( d(v) = 1 \)

  That’s all you need to prove almost-sure termination!
The symmetric-in-the-limit random walk

Consider the following program:

```plaintext
while (x > 0) { p := x/(2*x+1) ; x := x-1 [p] x := x+1 }
```
The symmetric-in-the-limit random walk

Consider the following program:

```c
while (x > 0) { p := x/(2*x+1) ; x := x-1 [p] x := x+1 }
```

Witnesses of almost-sure termination:

- \( V = H_x \), where \( H_k \) is the \(k\)-th Harmonic number \( 1 + \frac{1}{2} + \ldots + \frac{1}{k} \)
- \( p(v) = \frac{1}{3} \) and \( d(v) = \begin{cases} \frac{1}{k} & \text{if } v > 0 \text{ and } H_{k-1} < v \leq H_k \\ 1 & \text{if } v = 0 \end{cases} \)
The symmetric-in-the-limit random walk

Consider the following program:

```
while (x > 0) { p := x/(2*x+1) ; x := x-1 [p] x := x+1 }
```

Witnesses of almost-sure termination:

- $V = H_x$, where $H_k$ is $k$-th Harmonic number $1 + \frac{1}{2} + \ldots + \frac{1}{k}$
- $p(v) = \frac{1}{3}$ and $d(v) = \begin{cases} 
\frac{1}{k} & \text{if } v > 0 \text{ and } H_{k-1} < v \leq H_k \\
1 & \text{if } v = 0
\end{cases}$
Overview

1. Introduction
2. Probabilistic guarded command language
3. Predicate transformers
4. Termination
5. Synthesizing loop invariants
6. Recursion
7. Epilogue
Overview

1. Introduction
2. Probabilistic guarded command language
3. Predicate transformers
4. Termination
5. Synthesizing loop invariants
6. Recursion
7. Epilogue
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 $X - Y$
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 (!flip) { // take a sample of X to increase x
        (x++ [p] flip := true);
    }
    flip := false;
    while (!flip) { // take a sample of Y to decrease x
        (x-- [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++ [p] f := 1);
    }
    f := 0;
    while (f = 0) {
        (x-- [q] f := 1);
    }
    return x;
}
```

Using loop invariant synthesis:
Both programs are equivalent for any \( q \) with \( q = \frac{1}{2} - p \).
Program equivalence

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

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

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

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

Using loop invariant synthesis:

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

inspired by [Colón et al. 2002]

1. Speculatively annotate a while-loop 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., by SMT solving).
Soundness and completeness

For any linear probabilistic program and linear expectations this method will find all parameter solutions that make the template valid, and no others.
**Prinsys tool**: Probabilistic Invariants Synthesis

moves.rwth-aachen.de/prinsys
Program equivalence
Program equivalence

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

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

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

\[
\text{int } X\text{minY1(float } p, q){}
\text{ int } x, f := 0, 0;
\text{ while (f = 0) }{}
\text{ (x++ [p] f := 1);}{}
\text{ f := 0;}
\text{ while (f = 0) }{}
\text{ (x-- [q] f := 1);}{}
\text{ return x;}
\]

\[
\text{int } X\text{minY2(float } p, q){}
\text{ int } x, f := 0, 0;
\text{ if (f = 0) }{}
\text{ (f := 0 [0.5] f := 1);}{}
\text{ else }{}
\text{ f := 0;}
\text{ while (f = 0) }{}
\text{ (x++ [p] f := 1);}{}
\text{ x--;}{}
\text{ (skip [q] f := 1);}{}
\text{ return x;}
\]

Expected value of \(x\) is \(\frac{p}{1-p} - \frac{q}{1-q}\) and \(\frac{p}{2(1-p)} - \frac{1}{2(1-q)}\).
Overview

1 Introduction

2 Probabilistic guarded command language

3 Predicate transformers

4 Termination

5 Synthesizing loop invariants

6 Recursion

7 Epilogue
Overview

1. Introduction
2. Probabilistic guarded command language
3. Predicate transformers
4. Termination
5. Synthesizing loop invariants
6. Recursion
7. Epilogue
Recursion

Q: What is the probability that recursive program \texttt{call P} terminates?

\begin{verbatim}
P :: skip [0.5] { call P; call P; call P }
\end{verbatim}
The semantics of recursive procedures is the limit of their $n$-th inlining:

\[
\begin{align*}
\text{call}_0^D P &= \text{abort} \\
\text{call}_{n+1}^D P &= D(P)[\text{call } P := \text{call}_n^D P]
\end{align*}
\]

\[
wp(\text{call } P, f)[D] = \sup_n wp(\text{call}_n^D P, f)
\]

where $D$ is the process declaration and $D(P)$ the body of $P$. 

Recursion
The semantics of recursive procedures is the limit of their $n$-th inlining:

$$\text{call}_0^D P = \text{abort}$$
$$\text{call}_{n+1}^D P = D(P)[\text{call } P := \text{call}_n^D P]$$

$$\text{wp}(\text{call } P, f)[D] = \sup_n \text{wp}(\text{call}_{n}^D P, f)$$

where $D$ is the process declaration and $D(P)$ the body of $P$.

This corresponds to the fixed point of a (higher order) environment transformer.
Pushdown Markov chains

\[ \{ \text{skip}^1 \} \left[ \frac{1}{2} \right]^2 \{ \text{call } P^3; \text{ call } P^4; \text{ call } P^5 \} \]
$W_p = \text{expected rewards in pushdown MCs}$

For recursive program $P$ and post-expectation $f$:

$wp(P, f)$ for input $s$ equals the expected reward (that depends on $f$) to reach a terminal state in the pushdown MC $\llbracket P \rrbracket$ when starting with $s$.

---

$^4$ see [Brazdil, Esparza, Kiefer, Kucera, FMSD 2013].
Predictable Probabilistic Programming

Recursion

\[ W_p = \text{expected rewards in pushdown MCs} \]

For recursive program \( P \) and post-expectation \( f \):

\[ wp(P, f) \]

for input \( s \) equals the expected reward (that depends on \( f \)) to reach a terminal state in the pushdown MC \( \llbracket P \rrbracket \) when starting with \( s \).

Checking expected rewards in finite-control pushdown MCs is decidable.\(^4\)

---

\(^4\) see [Brazdil, Esparza, Kiefer, Kucera, FMSD 2013].
Proof rules for recursion

Standard proof rule for recursion:

\[
wp(\text{call } P, f) \leq g \text{ derives } wp(D(P), f) \leq g
\]

\[
wp(\text{call } P, f)[D] \leq g
\]

call \( P \) satisfies \( f, g \) if \( P \)'s body satisfies it,
assuming the recursive calls in \( P \)'s body do so too.
Proof rules for recursion

Standard proof rule for recursion:

\[
\begin{align*}
wp(\text{call } P, f) & \leq g \quad \text{derives} \quad wp(D(P), f) \leq g \\
wp(\text{call } P, f)[D] & \leq g
\end{align*}
\]

call \( P \) satisfies \( f, g \) if \( P \)'s body satisfies it, assuming the recursive calls in \( P \)'s body do so too.

Proof rule for obtaining two-sided bounds given \( \ell_0 = 0 \) and \( u_0 = 0 \):

\[
\begin{align*}
\ell_n & \leq wp(\text{call } P, f) \leq u_n \quad \text{derives} \quad \ell_{n+1} \leq wp(D(P), f) \leq u_{n+1} \\
\sup_n \ell_n & \leq wp(\text{call } P, f)[D] \leq \sup_n u_n
\end{align*}
\]
The golden ratio

Extension with proof rules allows to show e.g.,

\[
P :: \text{skip} \ [0.5] \ \{ \ \text{call} \ P; \ \text{call} \ P; \ \text{call} \ P \ \}\
\]

terminates with probability \( \frac{\sqrt{5} - 1}{2} = \frac{1}{\phi} = \phi \)
The golden ratio

Extension with proof rules allows to show e.g.,

\[
P :: \text{skip } [0.5] \{ \text{call } P; \text{call } P; \text{call } P \}
\]

terminates with probability \( \frac{\sqrt{5} - 1}{2} = \frac{1}{\phi} = \varphi \)

Or: apply to reason about Sherwood variants of binary search, quick sort etc.
\[
wp[\text{call } P](1) \leq \varphi \iff \wp[\mathcal{D}(P_{\text{rec}_3})](1) \leq \varphi
\]

\[
\begin{align*}
wp[\mathcal{D}(P_{\text{rec}_3})](1) &= \{\text{def. of } wp\} \\
&= \frac{1}{2} \cdot wp[\text{skip}](1) + \frac{1}{2} \cdot wp[\text{call } P_{\text{rec}_3}; \text{ call } P_{\text{rec}_3}; \text{ call } P_{\text{rec}_3}](1) \\
&= \{\text{def. of } wp\} \\
&= \frac{1}{2} + \frac{1}{2} \cdot wp[\text{call } P_{\text{rec}_3}; \text{ call } P_{\text{rec}_3}](wp[\text{call } P_{\text{rec}_3}](1)) \\
&= \{\text{assumption, monot. of } wp\} \\
&= \frac{1}{2} + \frac{1}{2} \varphi \cdot wp[\text{call } P_{\text{rec}_3}](wp[\text{call } P_{\text{rec}_3}](1)) \\
&= \{\text{assumption, monot. of } wp\} \\
&= \frac{1}{2} + \frac{1}{2} \varphi \cdot wp[\text{call } P_{\text{rec}_3}](\varphi) \\
&= \{\text{scalab. of } wp\} \\
&= \frac{1}{2} + \frac{1}{2} \varphi^2 \cdot wp[\text{call } P_{\text{rec}_3}](1) \\
&= \{\text{assumption, monot. of } wp\} \\
&= \frac{1}{2} + \frac{1}{2} \varphi^3 \\
&= \{\text{algebra}\}
\end{align*}
\]
Overview

1. Introduction
2. Probabilistic guarded command language
3. Predicate transformers
4. Termination
5. Synthesizing loop invariants
6. Recursion
7. Epilogue
Perspective

“There are several reasons why probabilistic programming could prove to be revolutionary for machine intelligence and scientific modelling.”

Why? Probabilistic programming

1. ... obviates the need to manually provide inference methods
2. ... enables rapid prototyping
3. ... clearly separates the model and the inference procedures
Epilogue

Take-home messages

▶ wp-reasoning provides exact reasoning at program code level
▶ Termination has several nuances and is hard
▶ Excellent playground for formal verification
Epilogue

Take-home messages

- \textit{wp-reasoning} provides exact reasoning at program code level
- Termination has several nuances and is hard
- Excellent playground for formal verification

Extensions

- Expected run-time analysis
- Bounded model checking
- Link to Bayesian networks
- Invariant synthesis
Epilogue

Take-home messages

- wp-reasoning provides exact reasoning at program code level
- Termination has several nuances and is hard
- Excellent playground for formal verification

Extensions

- Expected run-time analysis
- Bounded model checking
- Link to Bayesian networks
- Invariant synthesis

Grazie!
Further reading

- JPK, A. McIver, L. Meinicke, and C. Morgan.  
- F. Gretz, JPK, and A. McIver.  
- F. Olmedo *et al.*  
  *Conditioning in probabilistic programming.* ACM TOPLAS 2017.
- B. Kaminski, JPK.  
  *On the hardness of almost-sure termination.* MFCS 2015.
- B. Kaminski, JPK, C. Matheja, and F. Olmedo.  
- F. Olmedo, B. Kaminski, JPK, C. Matheja.  
  *Reasoning about recursive probabilistic programs.* LICS 2016.
- A. McIver, C. Morgan, B. Kaminski, JPK.  