Predictable Probabilistic Programming

by Deductive Verification

Joost-Pieter Katoen

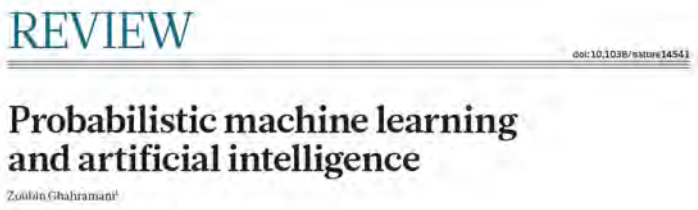
Software Modeling and Verification Chair | RWTH AACHEN UNIVERSITY

Workshop on Software Correctness and Reliability, ETH Zurich, 2017

Overview

nature Perspective

“There are several reasons why probabilistic programming could prove to be revolutionary for machine intelligence and scientific modelling.”[1]


REVIEW

doi:10.1038/nature14541

Probabilistic machine learning and artificial intelligence

Zoubin Ghahramani[1]

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

Probabilistic programs

What?

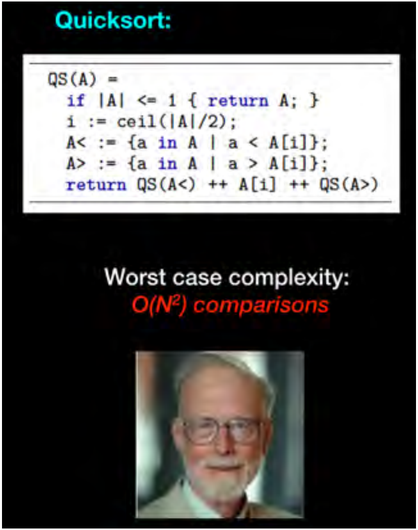
They are programs with random assignments and conditioning

Why?

- Random assignments: to describe randomised algorithms
- Conditioning: to describe stochastic decision making

Sorting by flipping coins

Sorting by flipping coins



Sorting by flipping coins

Quicksort:

```
QS(A) =
  if |A| <= 1 { return A; }
  i := ceil(|A|/2);
  A< := {a in A | a < A[i]};
  A> := {a in A | a > A[i]};
  return QS(A<) ++ A[i] ++ QS(A>)
```

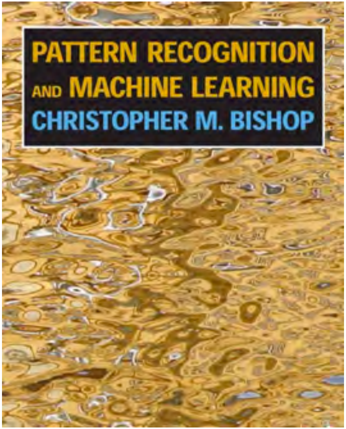
Worst case complexity:
$O(N^2)$ comparisons

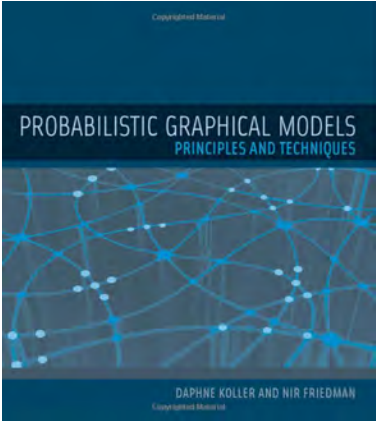
Randomised Quicksort:

```
rQS(A) =
  if |A| <= 1 { return A; }
  i := Unif[1...|A|];
  A< := {a in A | a < A[i]};
  A> := {a in A | a > A[i]};
  return rQS(A<) ++ A[i] ++ rQS(A>)
```

Worst case complexity:
O(N log N) expected comparisons

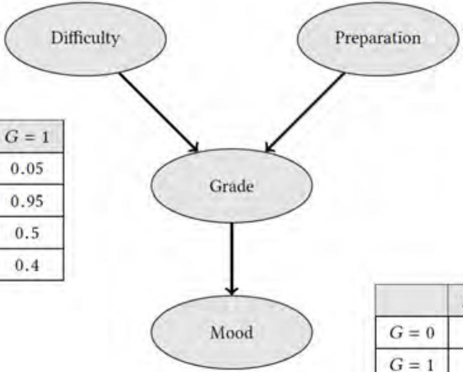
Probabilistic graphical models





Student's mood after an exam

$D = 0$	$D = 1$
0.6	0.4



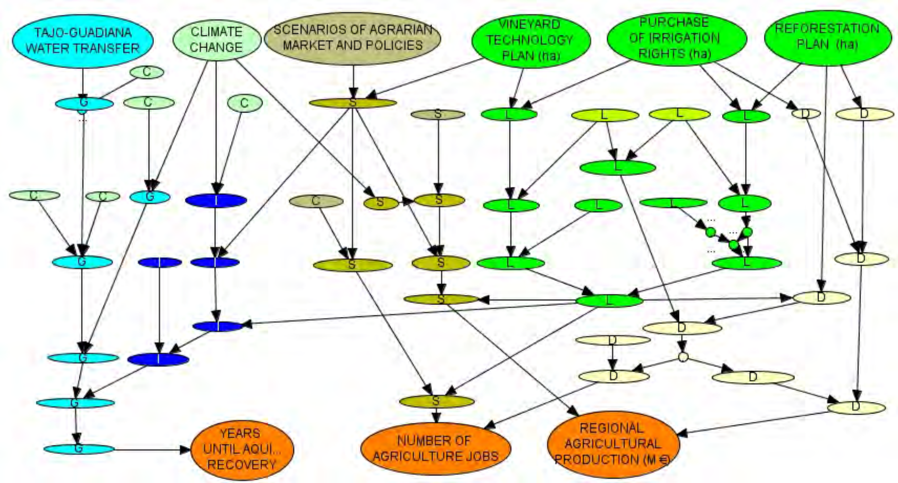
$P = 0$	$P = 1$
0.7	0.3

	$G = 0$	$G = 1$
$D = 0, P = 0$	0.95	0.05
$D = 1, P = 1$	0.05	0.95
$D = 0, P = 1$	0.5	0.5
$D = 1, P = 0$	0.6	0.4

	$M = 0$	$M = 1$
$G = 0$	0.9	0.1
$G = 1$	0.3	0.7

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

[Daniel Roy, 2011][a]

"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.

Rethinking the Bayesian approach

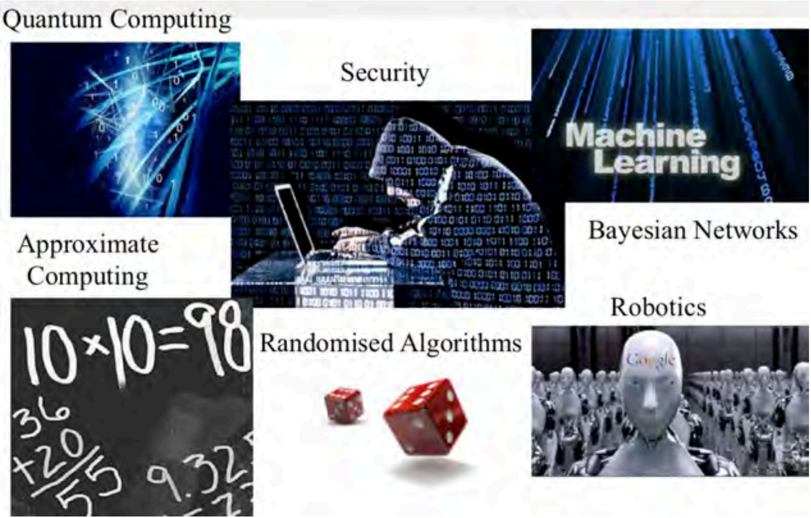
[Daniel Roy, 2011][a]

"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"

[a] 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
Microsoft Research

Roadmap

Overview

Probabilistic GCL

Kozen

McIver

Morgan

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

Let's start simple

```
x := 0 [0.5] x := 1;
y := -1 [0.5] y := 0
```

This program admits four runs and yields the outcome:

$$Pr[x=0, y=0] = Pr[x=0, y=-1] = Pr[x=1, y=0] = Pr[x=1, y=-1] = 1/4$$

A loopy program

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

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

The loopy program models a geometric distribution with parameter p.

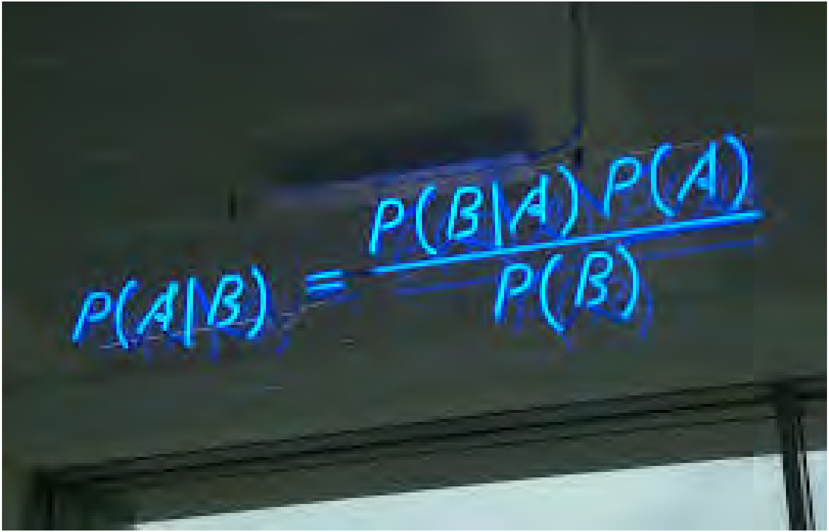
$$Pr[i = N] \;=\; (1-p)^{N-1} \cdot p \quad \text{for } N > 0$$

On termination

```
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



Let's start simple

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

Let's start simple

```
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:

$$Pr[x=0, y=0] = Pr[x=1, y=-1] = 1/2$$

Let's start simple

```
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:

$$Pr[x=0, y=0] = Pr[x=1, y=-1] = 1/2$$

Observations thus normalize the probability of the "feasible" program runs

A loopy program

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

```
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:

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

The feasible program runs have a probability $\sum_{N \geq 0} (1-p)^{2N} \cdot p \; = \; \dfrac{1}{2-p}$

A loopy program

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

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

The feasible program runs have a probability $\sum_{N \geq 0} (1-p)^{2N} \cdot p \;=\; \frac{1}{2-p}$

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

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 **[Dijkstra 1975]**

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 [Dijkstra 1975]

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.

Weakest preconditions

Weakest precondition [Dijkstra 1975]

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 preconditions

Weakest precondition [Dijkstra 1975]

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`

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

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

μ 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 **[McIver & Morgan 2004]**

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

[2] $\neq$ 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] $\neq$ 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 $f(s) \leq g(s)$.

An expectation transformer is a total function between two expectations.

[2] $\neq$ 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 $f(s) \leq g(s)$.

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.

[2] $\neq$ 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 $f(s) \leq g(s)$.

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] $\neq$ 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 $f(s) \leq g(s)$.

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] $\neq$ 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`

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

Expectation transformer semantics of pGCL

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

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

Expectation transformer semantics of pGCL

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

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

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

Wp = conditional rewards in Markov chains

For program P, input s and expectation f:

$$\frac{wp(P, f)(s)}{wlp(P, \mathbf{1})(s)} \quad = \quad \mathrm{ER}^{[\![P]\!]}\left(s, \Diamond\langle sink\rangle \cap \neg\Diamond\langle \lightning \rangle\right)$$

The ratio of $wp(P, f)$ over $wlp(P, \mathbf{1})$ 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.

Wp = conditional rewards in Markov chains

For program P, input s and expectation f:

$$\frac{wp(P, f)(s)}{wlp(P, \mathbf{1})(s)} \quad = \quad \mathrm{ER}^{[\![P]\!]}\left(s, \Diamond\langle sink\rangle \cap \neg\Diamond\langle \lightning \rangle\right)$$

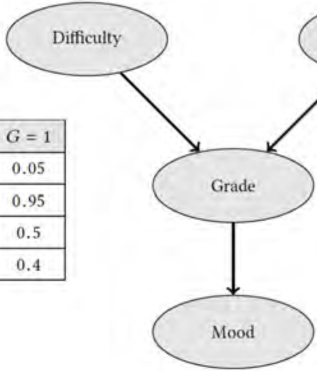
The ratio of $wp(P, f)$ over $wlp(P, \mathbf{1})$ for input s equals[3] the conditional expected reward to reach a successful terminal state $\langle sink \rangle$ while satisfying all observations in P's MC when starting with s.

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

$D=0$	$D=1$
0.6	0.4



$P=0$	$P=1$
0.7	0.3

	$G=0$	$G=1$
$D=0, P=0$	0.95	0.05
$D=1, P=1$	0.05	0.95
$D=0, P=1$	0.5	0.5
$D=1, P=0$	0.6	0.4

	$M=0$	$M=1$
$G=0$	0.9	0.1
$G=1$	0.3	0.7

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.:

```
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
}
```

	$G = 0$	$G = 1$
$D = 0, P = 0$	0.95	0.05
$D = 1, P = 1$	0.05	0.95
$D = 0, P = 1$	0.5	0.5
$D = 1, P = 0$	0.6	0.4

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

```
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 $\underline{v}$ for node (and input) v:

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

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

Soundness

Correctness of BN programs

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

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

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

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 $\underline{v}$ for node (and input) v:

$$wp(\texttt{prog}(B, obs), \bigwedge_{v \in V \setminus O} x_v = \underline{v}) \;=\; Pr\left(\bigwedge_{v \in V \setminus O} v = \underline{v} \;\middle|\; \bigwedge_{o \in O} o = \underline{o} \right)$$

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

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

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

Soundness

Correctness of BN programs

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

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

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

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

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

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

Overview

On termination

```
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

$\Longrightarrow$ almost-sure termination

Nuances of termination

...... certain termination

...... termination with probability one

$\Longrightarrow$ almost-sure termination

...... in an expected finite number of steps

$\Longrightarrow$ positive almost-sure termination

Positive almost-sure termination

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

```
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:

```
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:

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

Finite expected termination time

Compositionality

Consider the two probabilistic programs:

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

Finite expected termination time

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

Finite termination time

Compositionality

Consider the two probabilistic programs:

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

Finite expected termination time

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

Finite 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

$\underbrace{\text{Almost-sure termination}}_{\text{a single input}}$ is "as hard" as the $\underbrace{\text{universal halting problem}}_{\text{all inputs}}$

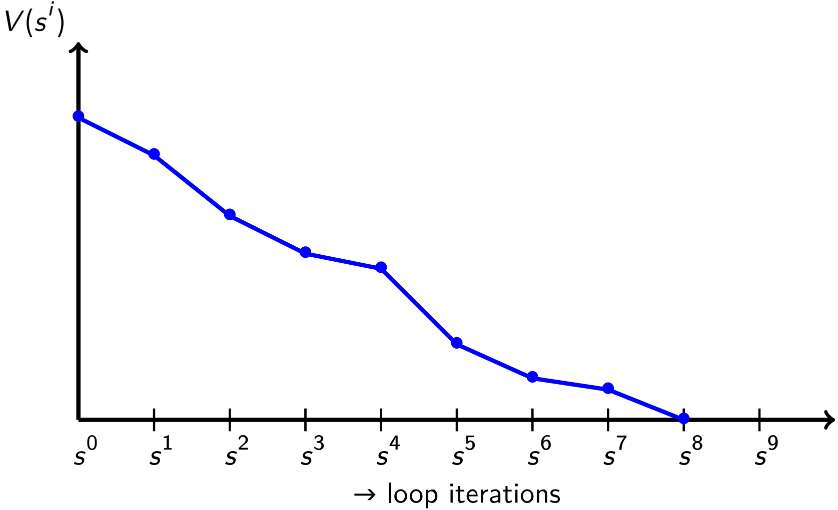
Termination is hard

Almost-sure termination is
"more undecidable"
than certain termination

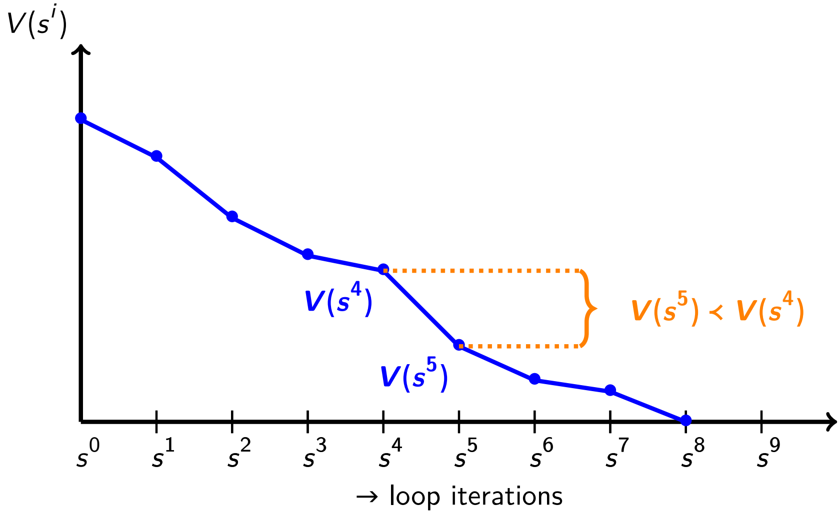
$\underbrace{\text{Almost-sure termination}}_{\text{a single input}}$ is "as hard" as the $\underbrace{\text{universal halting problem}}_{\text{all inputs}}$

Positive almost-sure termination is
"one degree more undecidable"
than almost-sure termination

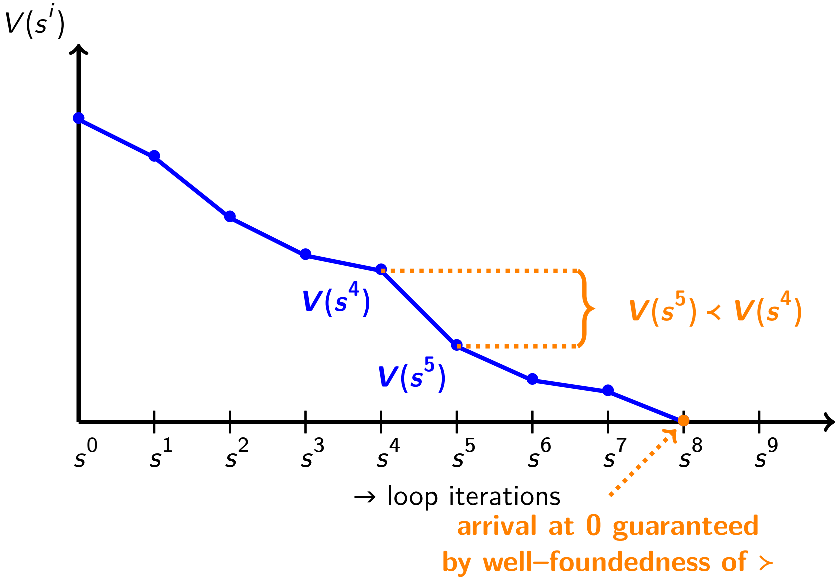
Termination proofs: the classical case



Termination proofs: the classical case



Termination proofs: the classical case



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)
}
```

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:

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

A loop iteration decreases x by one with probability $1/2$

This observation is enough to witness almost-sure termination!

Proving almost-sure termination

Goal: prove a.s.–termination of `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 $V(s) = 0$

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

Proving almost-sure termination

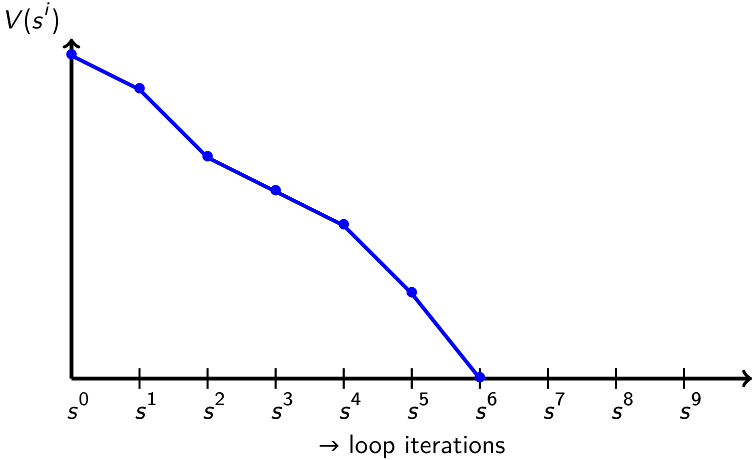
Goal: prove a.s.–termination of `while(G) P`, for probabilistic program `P`

Ingredients:

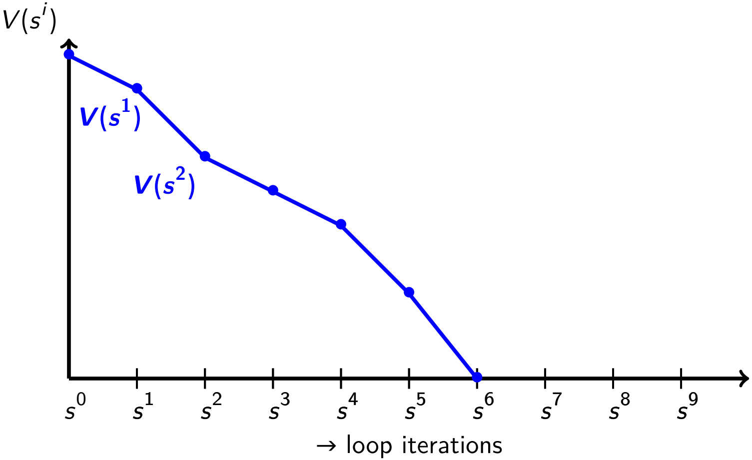
- A supermartingale V mapping states onto non-negative reals
  - Executing `P` on a state s satisfying `G` does not increase V's expected value
  - Loop iteration ceases if $V(s) = 0$
- and a progress condition: on each loop iteration
  - V's value v decreases by $\geq d(v)$ with probability $\geq p(v)$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

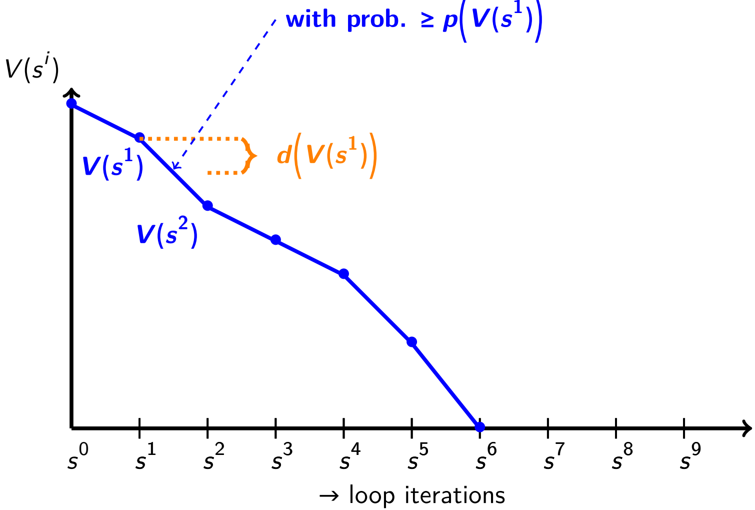
Proving almost-sure termination



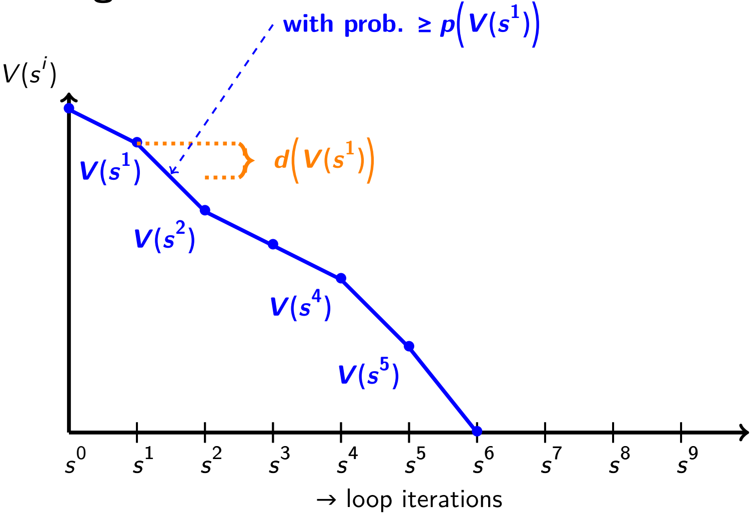
Proving almost-sure termination



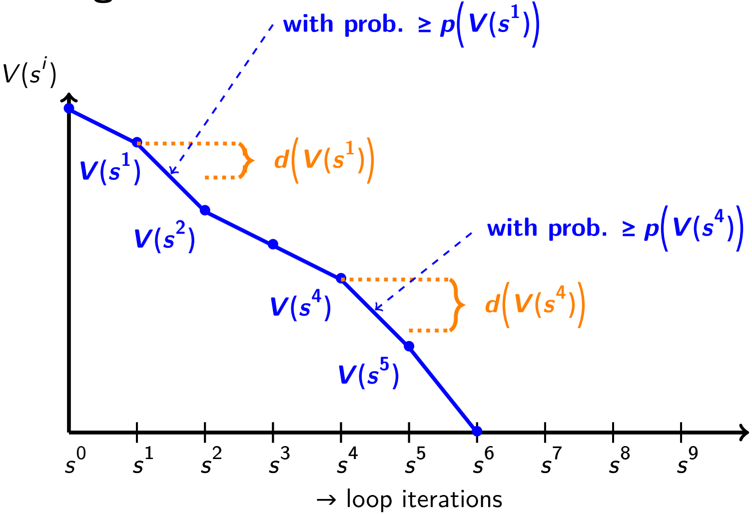
Proving almost-sure termination



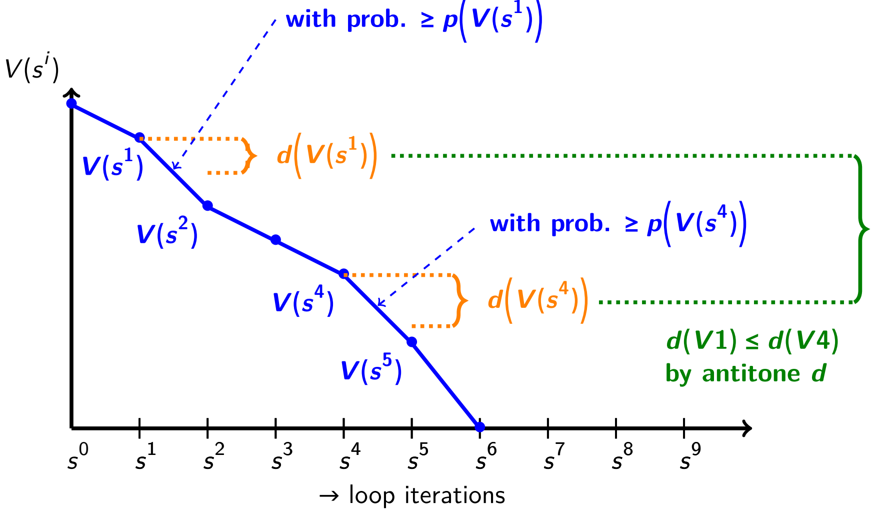
Proving almost-sure termination



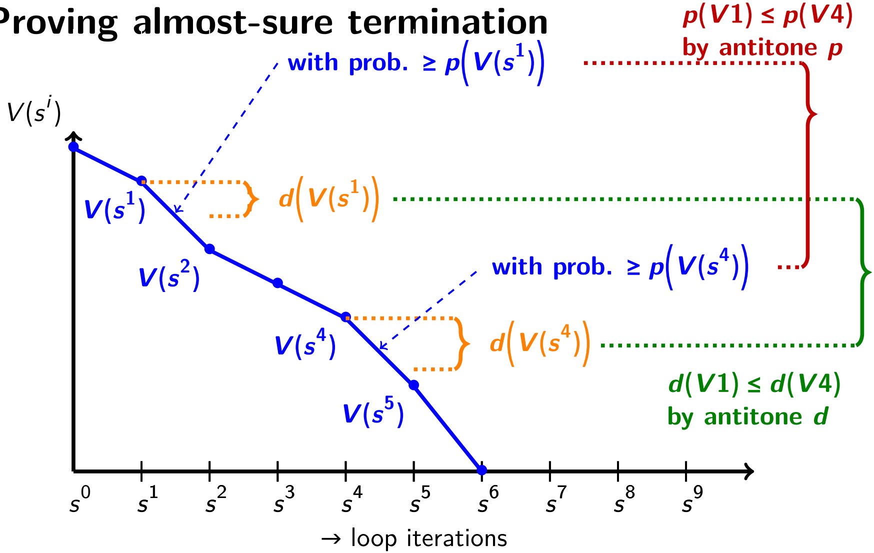
Proving almost-sure termination



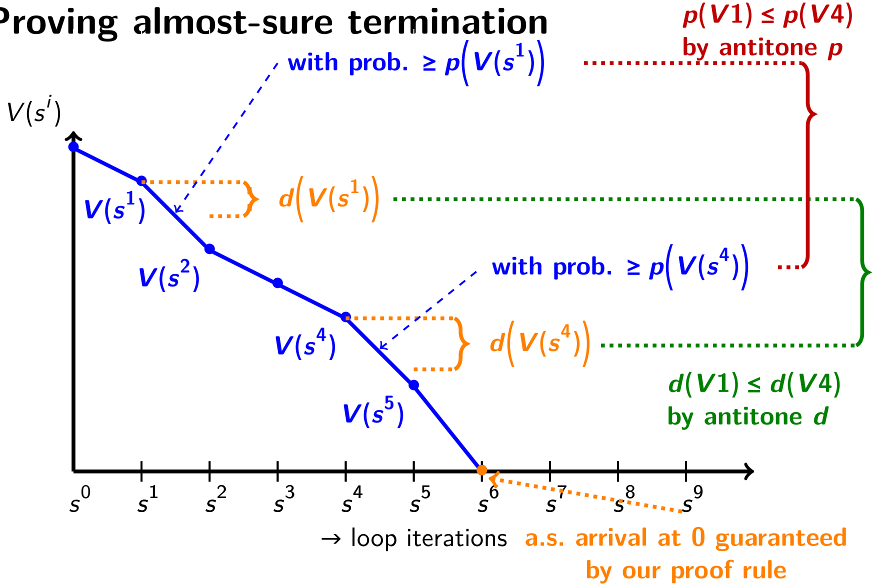
Proving almost-sure termination



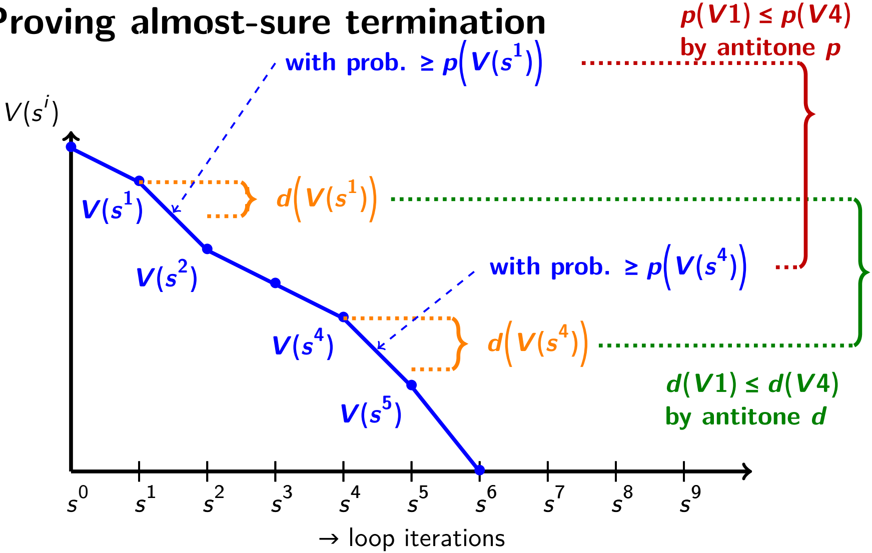
Proving almost-sure termination



Proving almost-sure termination



Proving almost-sure termination



The closer to termination, the more V decreases and this becomes more likely

The symmetric random walk

- Recall the program:

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

The symmetric random walk

- Recall the program:

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

- Witnesses of almost-sure termination:
  - $V = x$
  - $p(v) = 1/2$ 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:
  - $V = x$
  - $p(v) = 1/2$ 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:

```
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:

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

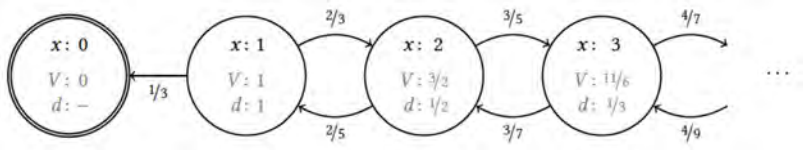
- Witnesses of almost-sure termination:
  - $V = H_x$, where H_k is k-th Harmonic number $1 + 1/2 + \ldots + 1/k$
  - $p(v) = 1/3$ and $d(v) = \begin{cases} 1/k & \text{if } v > 0 \text{ and } H_{k-1} < v \leq H_k \\ 1 & \text{if } v = 0 \end{cases}$

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 $1 + 1/2 + \ldots + 1/k$
  - $p(v) = 1/3$ and $d(v) = \begin{cases} 1/k & \text{if } v > 0 \text{ and } H_{k-1} < v \leq H_k \\ 1 & \text{if } v = 0 \end{cases}$



Overview

Overview

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 $X - Y$

```
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

```
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;
}
```

Program equivalence

```
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;
}
```

```
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;
}
```

Program equivalence

```
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;
}
```

```
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:

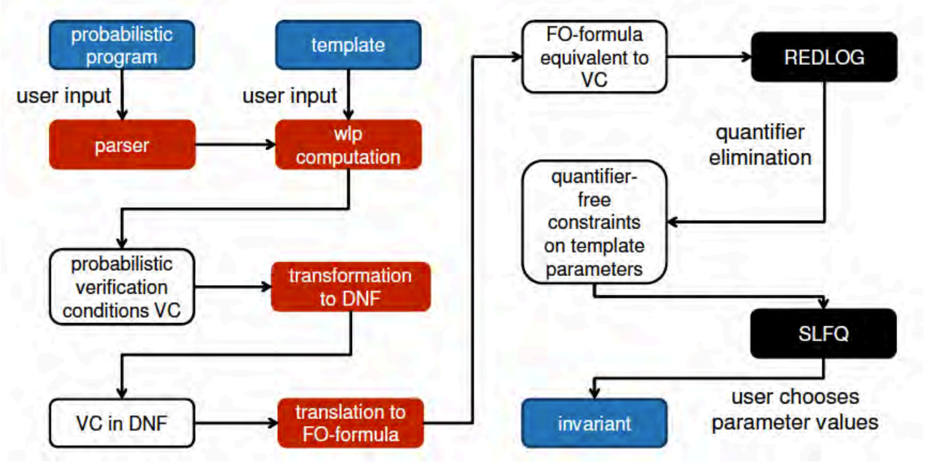
$$[\alpha_1 \cdot x_1 + \ldots + \alpha_n \cdot x_n + \alpha_{n+1} \ll 0] \cdot (\beta_1 \cdot x_1 + \ldots + \beta_n \cdot x_n + \beta_{n+1})$$

   with real parameters α_i, β_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

```
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;
}
```

```
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

```
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;
}
```

```
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;
}
```

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

Overview

Overview

Recursion

Q: What is the probability that recursive program `call P` terminates?

```
P :: skip [0.5] { call P; call P; call P }
```

Recursion

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

$$\begin{aligned}
\texttt{call}_0^D\ P &= \texttt{abort} \\
\texttt{call}_{n+1}^D P &= D(P)[\texttt{call}\ P := \texttt{call}_n^D\ P]
\end{aligned}$$

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

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:

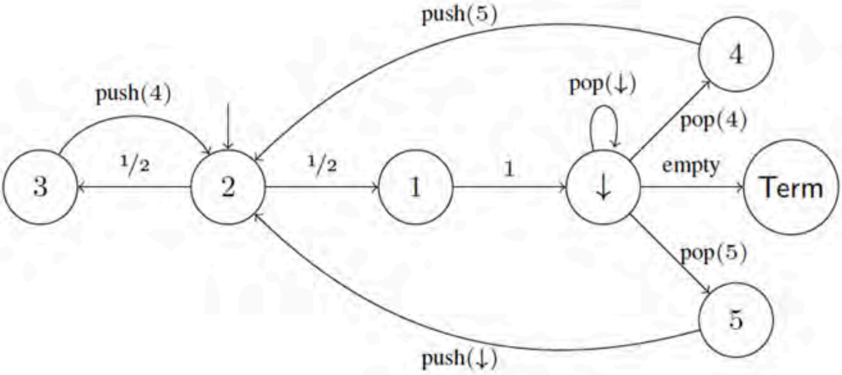
$$\begin{array}{rcl} \texttt{call}_0^D\ P & = & \texttt{abort} \\ \texttt{call}_{n+1}^D P & = & D(P)[\texttt{call}\ P := \texttt{call}_n^D\ P] \end{array}$$

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

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



$$\{\mathsf{skip}^1\}\ [1/2]^2\ \{\mathsf{call}\ P^3;\ \mathsf{call}\ P^4;\ \mathsf{call}\ P^5\ \}$$

Wp = 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 $[\![P]\!]$ when starting with s.

[4] see [Brazdil, Esparza, Kiefer, Kucera, FMSD 2013].

Wp = 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 $[\![P]\!]$ 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:

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

`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:

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

`call` P satisfies f, g if P's body satisfies it,

assuming the recursive calls in P's body do so too.

Proof rule for obtaining two-sided bounds given $\ell_0 = \mathbf{0}$ and $u_0 = \mathbf{0}$:

$$\frac{\ell_n \leq wp(\texttt{call}\, P, f) \leq u_n \text{ derives } \ell_{n+1} \leq wp(D(P), f) \leq u_{n+1}}{\sup_n \ell_n \;\leq\; wp(\texttt{call}\, P, f)[D] \;\leq\; \sup_n u_n}$$

The golden ratio

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

```
P :: skip [0.5] { call P; call P; call P }
```

terminates with probability $\dfrac{\sqrt{5}-1}{2} = \dfrac{1}{\phi} = \varphi$

The golden ratio

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

```
P :: skip [0.5] { call P; call P; call P }
```

terminates with probability $\dfrac{\sqrt{5}-1}{2} = \dfrac{1}{\phi} = \varphi$

Or: apply to reason about Sherwood variants of binary search, quick sort etc.

$$\mathsf{wp}[\mathsf{call}\ P](\mathbf{1}) \preceq \varphi \Vdash \mathsf{wp}[\mathcal{D}(P_{\mathsf{rec}_3})](\mathbf{1}) \preceq \varphi$$

$$
\begin{aligned}
& \mathsf{wp}[\mathcal{D}(P_{\mathsf{rec}_3})](\mathbf{1}) \\
= & \quad \{\text{def. of wp}\} \\
& \tfrac{1}{2} \cdot \mathsf{wp}[\mathsf{skip}](\mathbf{1}) + \tfrac{1}{2} \cdot \mathsf{wp}[\mathsf{call}\ P_{\mathsf{rec}_3};\ \mathsf{call}\ P_{\mathsf{rec}_3};\ \mathsf{call}\ P_{\mathsf{rec}_3}](\mathbf{1}) \\
= & \quad \{\text{def. of wp}\} \\
& \tfrac{1}{2} + \tfrac{1}{2} \cdot \mathsf{wp}[\mathsf{call}\ P_{\mathsf{rec}_3};\ \mathsf{call}\ P_{\mathsf{rec}_3}](\mathsf{wp}[\mathsf{call}\ P_{\mathsf{rec}_3}](\mathbf{1})) \\
\preceq & \quad \{\text{assumption, monot. of wp}\} \\
& \tfrac{1}{2} + \tfrac{1}{2} \cdot \mathsf{wp}[\mathsf{call}\ P_{\mathsf{rec}_3};\mathsf{call}\ P_{\mathsf{rec}_3}](\varphi) \\
= & \quad \{\text{def. of wp, scalab. of wp twice}\} \\
& \tfrac{1}{2} + \tfrac{1}{2}\,\varphi \cdot \mathsf{wp}[\mathsf{call}\ P_{\mathsf{rec}_3}](\mathsf{wp}[\mathsf{call}\ P_{\mathsf{rec}_3}](\mathbf{1})) \\
\preceq & \quad \{\text{assumption, monot. of wp}\} \\
& \tfrac{1}{2} + \tfrac{1}{2}\,\varphi \cdot \mathsf{wp}[\mathsf{call}\ P_{\mathsf{rec}_3}](\varphi) \\
= & \quad \{\text{scalab. of wp}\} \\
& \tfrac{1}{2} + \tfrac{1}{2}\,\varphi^2 \cdot \mathsf{wp}[\mathsf{call}\ P_{\mathsf{rec}_3}](\mathbf{1}) \\
\preceq & \quad \{\text{assumption, monot. of wp}\} \\
& \tfrac{1}{2} + \tfrac{1}{2}\,\varphi^3 \\
= & \quad \{\text{algebra}\} \\
& \varphi \qquad\qquad \triangle
\end{aligned}
$$

Overview

nature 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

- 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

Further reading

- JPK, A. McIver, L. Meinicke, and C. Morgan.
  Linear-invariant generation for probabilistic programs. SAS 2010.
- F. Gretz, JPK, and A. McIver.
  Operational versus wp-semantics for `pGCL`. J. on Performance Evaluation, 2014.
- 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.
  Expected run-time analysis of probabilistic programs . ESOP 2016.
- F. Olmedo, B. Kaminski, JPK, C. Matheja.
  Reasoning about recursive probabilistic programs. LICS 2016.
- A. McIver, C. Morgan, B. Kaminski, JPK.
  A new proof rule for almost-sure termination. POPL 2018.