# Deductive Verification of Probabilistic Programs

Joost-Pieter Katoen

ıuıagen der Programmıeıu g

# Probabilistic programs

Programs with random assignments and conditioning

- naturally code up randomised algorithms
- represent probabilistic graphical models beyond Bayesian networks
- model controllers for autonomous robots
- key to describe security mechanisms
- ......

Programming languages: R2, STAN, Pyro, PyMC, WebPPL, Fabular, ...

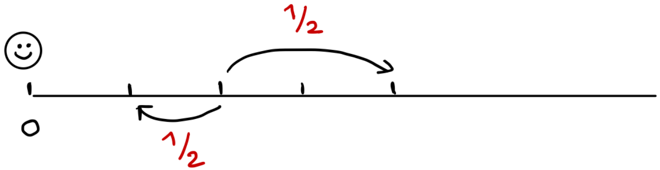
"Probabilistic programming aims to make probabilistic modeling and machine learning accessible to the programmer."[1]

[1] [Gordon, Henzinger, Nori and Rajamani, FOSE 2014]

# Probabilistic programs are hard to grasp

Does this program almost surely terminate? That is, is it AST?

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



# Probabilistic programs are hard to grasp

Does this program almost surely terminate? That is, is it AST?

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

If not, what is its probability to diverge?

# Even if all loops are bounded

[Flajolet *et al.*, 2009]

```
x := geometric(1/4);
y := geometric(1/4);
t := x+y;
t := t+1 [5/9] skip;
r := 1;
for i in 1..3 {
   s := iid(bernoulli(1/2), 2t);
   if (s != t) { r := 0 } else skip
}
```

```
s := 0;
for j := 1 to 2t {
    s++ [1/2] skip
}
```

# Even if all loops are bounded

```
x := geometric(1/4);
y := geometric(1/4);
t := x+y;
t := t+1 [5/9] skip;
r := 1;
for i in 1..3 {
   s := iid(bernoulli(1/2), 2t);
   if (s != t) { r := 0 } else skip
}
```

What is the probability that r equals one on termination?

# Positive almost-sure termination

$$E[X] = \sum_{k=1}^{\infty} \frac{1}{2^k} \cdot 2^k$$

```
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--
}
```

Finite termination time!

Expected runtime of these programs in sequence?

# Our objective

A powerful, simple proof calculus for probabilistic programs.

At the source code level.

No "descend" into the underlying probabilistic model.

Push automation as much as we can.

This is a true challenge: undecidability!

Typically "more undecidable" than deterministic programs

# Overview

# Expectation transformers

The set of expectations[1] (read: random variables):

$$\mathbb{E} = \left\{ f \mid f : \underbrace{\mathbb{S}}_{\text{states}} \rightarrow \mathbb{R}_{\geq 0} \cup \{ \infty \} \right\}$$

$$s \in \mathbb{S} : \quad \mathrm{Var} \longrightarrow \mathrm{Val}$$

[1] ≠ expectations in probability theory.

# Expectation transformers

The set of expectations[1] (read: random variables):

$$\mathbb{E} = \left\{ f \mid f : \underbrace{\mathbb{S}}_{\text{states}} \to \mathbb{R}_{\geq 0} \cup \{ \infty \} \right\}$$

$(\mathbb{E}, \sqsubseteq)$ is a complete lattice where $f \sqsubseteq g$ if and only if $\forall s \in \mathbb{S}.\, f(s) \leq g(s)$

[1] ≠ expectations in probability theory.

# Expectation transformers

The set of expectations[1] (read: random variables):

$$\mathbb{E} = \left\{ f \mid f : \underbrace{\mathbb{S}}_{\text{states}} \rightarrow \mathbb{R}_{\geq 0} \cup \{ \infty \} \right\}$$

$(\mathbb{E}, \sqsubseteq)$ is a complete lattice where $f \sqsubseteq g$ if and only if $\forall s \in \mathbb{S}.\, f(s) \leq g(s)$

The function $\Phi : \mathbb{E} \rightarrow \mathbb{E}$ is an expectation transformer

expectations are the quantitative analogue of predicates

[1] $\neq$ expectations in probability theory.

# Weakest pre-expectations

For prob. program $P$, let $wp[\![P]\!] : \mathbb{E} \to \mathbb{E}$ an expectation transformer

> $g = wp[\![P]\!](f)$ is $P$'s weakest pre-expectation w.r.t. post-expectation $f$ iff
>
> the expected value of $f$ after executing $P$ on input $s$ equals $g(s)$

Examples:

For $P$:: `x := x+5 [4/5] x := 10`, we have:

$$wp[\![P]\!](x) = \frac{4x}{5} + 6 \quad \text{and} \quad wp[\![P]\!]([x = 10]) = \frac{4 \cdot [x = 5] + 1}{5}$$

expected value of x — probability that x = 10

# Weakest pre-expectations

For prob. program $P$, let $wp[\![P]\!] : \mathbb{E} \to \mathbb{E}$ an expectation transformer

$g = wp[\![P]\!](f)$ is $P$'s weakest pre-expectation w.r.t. post-expectation $f$ iff

the expected value of $f$ after executing $P$ on input $s$ equals $g(s)$

$wp[\![P]\!]([true])$

Examples:

For $P$:: `x := x+5 [4/5] x := 10`, we have:

$$wp[\![P]\!](x) = \frac{4x}{5} + 6 \quad \text{and} \quad wp[\![P]\!]([x = 10]) = \frac{4 \cdot [x = 5] + 1}{5}$$

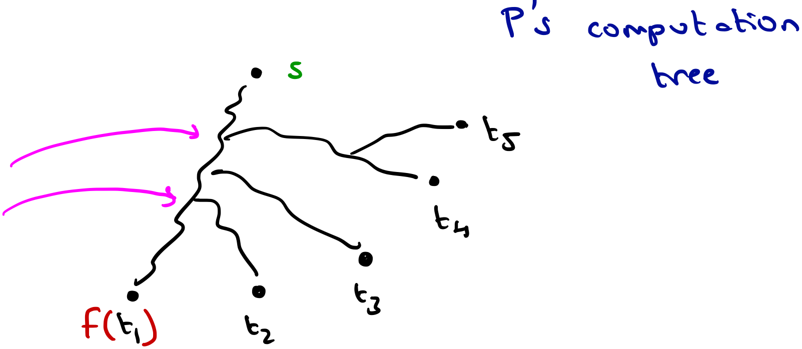
$wp[\![P]\!]([F])$ is the probability of predicate $F$ on $P$'s termination

# Kozen's duality theorem

If $\mu_P^s$ is the distribution over the final states obtained by running $P$ on the initial state $s$, then for post-expectation $f$:

$$\underbrace{\sum_{t \in \mathbb{S}} \mu_P^s(t) \cdot f(t)}_{\text{forward}} \;=\; \underbrace{wp[\![P]\!](f)(s)}_{\text{backward}}$$

Pictorially:




$x^2+y$

# Expectation transformer semantics

| Syntax probabilistic program $P$ | Semantics $wp[\![P]\!](f)$ |
|---|---|
| `skip` | $f$ |
| $x := E$ | $f[x := E]$ |
| $x :\approx \mu$ | $\lambda s. \int_{\mathbb{Q}} (\lambda v. f(s[x := v]))\, d\mu_s$ |
| $P; Q$ | $wp[\![P]\!](wp[\![Q]\!](f))$ |
| `if` $(\varphi)$ $P$ `else` $Q$ | $[\varphi] \cdot wp[\![P]\!](f) + [\neg\varphi] \cdot wp[\![Q]\!](f)$ |
| $P\,[p]\,Q$ | $p \cdot wp[\![P]\!](f) + (1-p) \cdot wp[\![Q]\!](f)$ |
| `while` $(\varphi)$ $\{P\}$ | $\text{lfp}\, X. \underbrace{(([\varphi] \cdot wp[\![P]\!](X)) + [\neg\varphi] \cdot f)}_{\text{loop characteristic function } \Phi_f(X)}$ |

# Examples

weakest pre-expectation: $\frac{\sqrt{5}-1}{2}$

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

post-expectation: **1**

```
x := geometric(1/4);
y := geometric(1/4);
t := x+y;
t := t+1 [5/9] skip;
r := 1;
for i in 1..3 {
 s := iid(bernoulli(1/2),2t);
 if (s != t) { r := 0 }
}
```

# Examples

weakest pre-expectation: $\frac{\sqrt{5}-1}{2}$

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

post-expectation: $\mathbf{1}$

weakest pre-expectation: $\frac{1}{\pi}$

```
x := geometric(1/4);
y := geometric(1/4);
t := x+y;
t := t+1 [5/9] skip;
r := 1;
for i in 1..3 {
 s := iid(bernoulli(1/2),2t);
 if (s != t) { r := 0 }
}
```

post-expectation: $[\mathbf{r} = \mathbf{1}]$

# Extensions of probabilistic wp

- ...... for recursion [LICS 2016]
- ...... for exact inference [TOPLAS 2018]
- ...... for continuous distributions [SETTS 2019]
- ...... for expected runtime analysis [JACM 2018]
- ...... for probabilistic separation logic [POPL 2019]
- ...... for weighted programs [OOPSLA 2022]
- ...... for amortised complexity analysis [POPL 2023]

# "How long does your program take on average?"

## EXPECTED RUNTIMES

Hanne Riis Nielson: Hoare Logic for Deterministic Runtimes (1984)

# Expected runtimes

Expected runtime of program $P$ on input $s$:

$$\sum_{i=1}^{\infty} i \cdot Pr\left( \begin{array}{c} \text{“}P \text{ terminates after} \\ i \text{ steps on input } s\text{”} \end{array} \right)$$

$ert[\![P]\!](t)(s)$ = expected runtime of $P$ on $s$ where $t$ is runtime after $P$

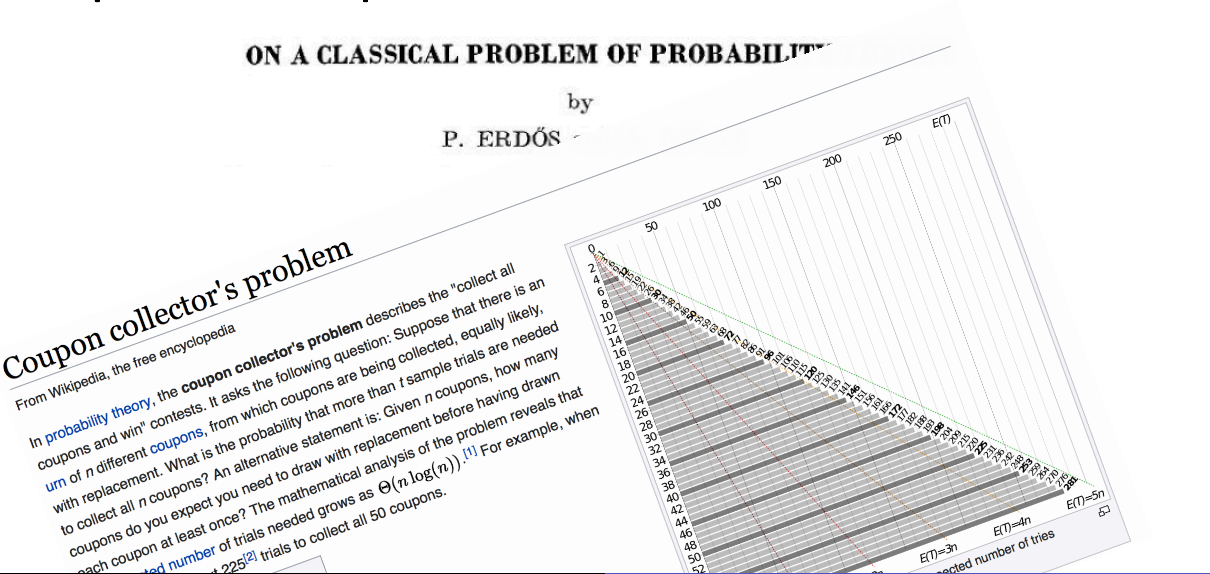
# Coupon collector's problem

ON A CLASSICAL PROBLEM OF PROBABILITY THEORY

by

P. ERDŐS and A. RÉNYI

# Coupon collector's problem



# Coupon collector's problem

```
cp := [0,...,0]; // no coupons yet
i := 1; // coupon to be collected next
x := 0: // number of coupons collected
while (x < N) {
   while (cp[i] != 0) {
       i := uniform(1..N) // next coupon
   }
   cp[i] := 1;  // coupon i obtained
   x++;  // one coupon less to go
}
```

The expected runtime of this program is in $\Theta(N \cdot \log N)$.

Can one formally derive such results by a syntax-directed program analysis?

# Some hurdles in runtime analysis

1. Programs may diverge despite having a finite expected runtime:

```
while (x > 0) { x-- [1/2] skip }
```

2. Expected runtimes are extremely sensitive

```
while (x > 0) { x-- [1/2-e] x++ } // -1/2 <= e <= 1/2
```

   - $e = 0$: almost-sure termination, infinite expected runtime
   - $e > 0$: not almost-sure termination, infinite expected runtime
   - $e < 0$: almost-sure termination, finite expected runtime (= PAST)

3. Having a finite expected time is not compositional

# Counterexample: why ghost code fails

```
while(true) { skip; x++ }
```

- Post: $x$, as seemingly x counts #loop iterations
- Characteristic function: $\Phi_x(Y) \;=\; Y(x \mapsto x+1)$
- Candidate upper bound: $I \;=\; \mathbf{0}$
- Induction: $\Phi_x(I) \;=\; \mathbf{0}(x \mapsto x+1) \;=\; \mathbf{0} \;=\; I \;\sqsubseteq\; I$
- By Park induction: $\Phi_x(I) \;\sqsubseteq\; I$ implies $wp[\![\text{loop}]\!](x) \;\sqsubseteq\; I$

We — wrongly — get runtime $\mathbf{0}$. wp is unsound for expected runtimes.

# Expected run-time transformer semantics

| Syntax $P$ | Runtime-semantics $ert[\![P]\!](f)$ |
| --- | --- |
| `skip` | $f$ |
| $x := E$ | $f[x := E]$ |
| $x :\approx \mu$ | $\lambda s. \int_{\mathbb{Q}} (\lambda v. f(s[x := v]))\, d\mu_s$ |
| $P; Q$ | $ert[\![P]\!](ert[\![Q]\!](f))$ |
| `if` $(\varphi)$ $P$ `else` $Q$ | $[\varphi] \cdot ert[\![P]\!](f) + [\neg\varphi] \cdot ert[\![Q]\!](f)$ |
| $P\,[p]\,Q$ | $p \cdot ert[\![P]\!](f) + (1-p) \cdot ert[\![Q]\!](f)$ |
| `while` $(\varphi)$ $\{P\}$ | $\mathrm{lfp}\, X. \underbrace{(([\varphi] \cdot ert[\![P]\!](X)) + [\neg\varphi] \cdot f)}_{\text{loop characteristic function } \Phi_f(X)}$ |

# Expected run-time transformer semantics

| Syntax $P$ | Runtime-semantics $ert[\![P]\!](f)$ |
|---|---|
| `skip` | $\mathbf{1} + f$ |
| $x := E$ | $\mathbf{1} + f[x := E]$ |
| $x :\approx \mu$ | $\mathbf{1} + \lambda s. \int_{\mathbb{Q}} (\lambda v. f(s[x := v]))\, d\mu_s$ |
| $P; Q$ | $ert[\![P]\!]\,(ert[\![Q]\!](f))$ |
| `if` $(\varphi)$ $P$ `else` $Q$ | $\mathbf{1} + [\varphi] \cdot ert[\![P]\!](f) + [\neg\varphi] \cdot ert[\![Q]\!](f)$ |
| $P\,[p]\,Q$ | $\mathbf{1} + p \cdot ert[\![P]\!](f) + (1-p) \cdot ert[\![Q]\!](f)$ |
| `while` $(\varphi)$ $\{P\}$ | $\text{lfp}\, X. \mathbf{1} + \underbrace{(([\varphi] \cdot ert[\![P]\!](X)) + [\neg\varphi] \cdot f)}_{\text{loop characteristic function } \Phi_f(X)}$ |

# Expected run-time transformer semantics

| Syntax $P$ | Runtime-semantics $ert[\![P]\!](f)$ |
|---|---|
| `skip` | $\mathbf{1} + f$ |
| $x := E$ | $\mathbf{1} + f[x := E]$ |
| $x :\approx \mu$ | $\mathbf{1} + \lambda s. \int_{\mathbb{Q}} (\lambda v. f(s[x := v]))\, d\mu_s$ |
| $P; Q$ | $ert[\![P]\!](ert[\![Q]\!](f))$ |
| `if` $(\varphi)$ $P$ `else` $Q$ | $\mathbf{1} + [\varphi] \cdot ert[\![P]\!](f) + [\neg\varphi] \cdot ert[\![Q]\!](f)$ |
| $P\,[p]\,Q$ | $\mathbf{1} + p \cdot ert[\![P]\!](f) + (1-p) \cdot ert[\![Q]\!](f)$ |
| `while` $(\varphi)$ $\{P\}$ | $\text{lfp}\, X.\, \mathbf{1} + \underbrace{(([\varphi] \cdot ert[\![P]\!](X)) + [\neg\varphi] \cdot f)}_{\text{loop characteristic function } \Phi_f(X)}$ |

Very simple, but/and sound!

# Proving PAST

The ert-transformer enables to prove

that a program is positively almost-surely terminating

in a compositional manner,

although PAST itself is not compositional.

# Relevance

- Expected runtime analysis of randomised algorithms
- Proving positive almost-sure termination
- Basis for amortised expected runtimes
- Generalised to expected runtimes of quantum programs
- Automated resource analysis of probabilistic programs
- ......

More details in the next parts ... ...

# Overview

# Loops

body

$$wp[\![\text{while }(\varphi)\,\{\,P\,\}]\!](f) \;=\; \text{lfp}\,X.\;\underbrace{\left([\varphi]\cdot wp[\![\text{body}]\!](X) \;+\; [\neg\varphi]\cdot f\right)}_{\text{loop characteristic function }\Phi_f(X)}$$

# Loops

body

$$wp[\![\text{while } (\varphi)\,\{\, P \,\}]\!](f) \;=\; \text{lfp}\, X.\ \underbrace{([\varphi]\cdot wp[\![\text{body}]\!](X) + [\neg\varphi]\cdot f)}_{\text{loop characteristic function } \Phi_f(X)}$$

- Function $\Phi_f : \mathbb{E} \rightarrow \mathbb{E}$ is Scott continuous on $(\mathbb{E}, \sqsubseteq)$
- By Kleene's fixed point theorem, it follows: $\text{lfp}\ \Phi_f \;=\; \sup_{n\in\mathbb{N}} \Phi_f^n(\mathbf{0})$

# Upper bounds

Recall:

$$wp[\![\text{while } (\varphi)\,\{\,\text{body}\,\}]\!](f) \;=\; \text{lfp}\, X.\; \underbrace{([\varphi]\cdot wp[\![\text{body}]\!](X) \;+\; [\neg\varphi]\cdot f)}_{\Phi_f(X)}$$

Park induction:

$$\underbrace{\Phi_f(I) \sqsubseteq I}_{\text{an ``upper'' invariant}} \quad \text{implies} \quad \underbrace{wp[\![\text{while}(\varphi)\{\text{body}\}]\!](f)}_{\text{lfp}\,\Phi_f} \sqsubseteq I$$

# Upper bounds

Recall:

$$wp[\![\text{while}\,(\varphi)\,\{\,\text{body}\,\}]\!](f) \;=\; \text{lfp}\, X.\; \underbrace{([\varphi]\cdot wp[\![\text{body}]\!](X) \,+\, [\neg\varphi]\cdot f)}_{\Phi_f(X)}$$

Park induction:

$$\underbrace{\Phi_f(I) \sqsubseteq I}_{\text{an ``upper'' invariant}} \qquad \text{implies} \qquad \underbrace{wp[\![\text{while}(\varphi)\{\text{body}\}]\!](f)}_{\text{lfp}\,\Phi_f} \sqsubseteq I$$

Example: `while(c = 0) { x++ [p] c := 1 }`

$I \;=\; x + [c = 0]\cdot \frac{p}{1-p}$ is an "upper"-invariant w.r.t. $f = x$

# Lower bounds for PAST loops

**[Hark, K., *et al.*, POPL 2020]**

$$\left( I \sqsubseteq \Phi_f(I) \;\wedge\; \text{side conditions} \right) \quad \textit{implies} \quad I \sqsubseteq \mathrm{lfp}\, \Phi_f$$

# Lower bounds for PAST loops

**[Hark, K., *et al.*, POPL 2020]**

$$\big(I \sqsubseteq \Phi_f(I) \;\wedge\; \text{side conditions}\big) \quad \textit{implies} \quad I \sqsubseteq \operatorname{lfp} \Phi_f$$

where the side conditions:

PAST

1. while($\varphi$){body} terminates in finite expected time, and

2. for any $s \models \varphi$, $\underbrace{wp[\![\text{body}]\!](|I(s) - I|)(s) \;\leq\; c}_{\text{conditional difference boundedness}}$ for some $c \in \mathbb{R}_{\geq 0}$

# Lower bounds for PAST loops

[Hark, K., *et al.*, POPL 2020]

$$\big(I \sqsubseteq \Phi_f(I) \;\wedge\; \text{side conditions}\big) \quad \textit{implies} \quad I \sqsubseteq \mathrm{lfp}\,\Phi_f$$

where the side conditions :

1. while($\varphi$){body} terminates in finite expected time, and

2. for any $s \models \varphi$, $\underbrace{wp[\![\text{body}]\!](|I(s) - I|)(s) \;\leq\; c}_{\text{conditional difference boundedness}}$ for some $c \in \mathbb{R}_{\geq 0}$

Example. `while`$(c = 0)\{\, x{+}{+}\, [p]\, c := 1 \,\}$ is PAST, and

$$I \;=\; x + [c = 0] \cdot \frac{p}{1-p} \text{ is a "lower"-invariant w.r.t. } f = x$$

# Proving PAST

**[Chakarov & Sankaranarayan, CAV 2013]**

Consider the loop $\texttt{while}(\varphi)\{\, body \,\}$ and let:

$$V : \mathbb{S} \rightarrow \mathbb{R} \quad \text{with} \qquad [V \leq 0] = [\neg\varphi]$$

That is, $V \leq 0$ indicates termination.

If for some $\varepsilon > 0$:

$$\underbrace{[\varphi] \cdot wp[\![body]\!](V) \;\leq\; V - \varepsilon}_{\text{expected value of } V \text{ decreases by at least } \varepsilon}$$

Then:

the loop is PAST

# Example: symmetric 1D random walk

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

# Lower bounds on AST

**[McIver, K., et al., POPL 2018]**

Consider the loop $\texttt{while}(\varphi)\{\, \mathit{body} \,\}$ and let:

- $V : \mathbb{S} \to \mathbb{R}_{\geq 0}$ with $[\, V = 0] = [\neg \varphi]$ — $V = 0$ indicates termination
- $p : \mathbb{R}_{\geq 0} \to (0, 1]$ antitone — probability
- $d : \mathbb{R}_{\geq 0} \to \mathbb{R}_{>0}$ antitone — decrease

# Lower bounds on AST

[McIver, K., et al., POPL 2018]

Consider the loop $\texttt{while}(\varphi)\{\, body \,\}$ and let:

- $V : \mathbb{S} \to \mathbb{R}_{\geq 0}$ with $[V = 0] = [\neg\varphi]$ — $V = 0$ indicates termination
- $p : \mathbb{R}_{\geq 0} \to (0, 1]$ antitone — probability
- $d : \mathbb{R}_{\geq 0} \to \mathbb{R}_{>0}$ antitone — decrease

If:

$$\underbrace{[\varphi] \cdot wp[\![body]\!](V) \;\leq\; V}_{\text{expected value of } V \text{ does not increase}}$$

and

$$\underbrace{[\varphi] \cdot (p \circ V) \;\leq\; \lambda s.\, wp[\![body]\!]\,(|V \leq V(s) - d(V(s))|)\,(s)}_{\text{with at least prob. } p,\ V \text{ decreases at least by } d}$$

Then:

$$wp[\![\text{loop}]\!](\mathbf{1}) \;=\; \mathbf{1} \quad \text{i.e., loop is AST}$$

# Example: symmetric 1D random walk

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

- Terminates almost surely, but with infinite expected runtime
- Witness of almost-sure termination:
  - $V = x$
  - $p = 1/2$ and
  - $d = 1$

# Example: symmetric 1D random walk

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

- Terminates almost surely, but with infinite expected runtime
- Witness of almost-sure termination:
  - $V = x$
  - $p = 1/2$ and
  - $d = 1$

can be fully automated (Amber)

That's all you need to prove almost-sure termination!

# Overview

# Relative complete verification

## Ordinary Programs

$F \in$ FO-Arithmetic

implies

$wp[\![P]\!](F) \in$ FO-Arithmetic

$$G \Longrightarrow wp[\![P]\!](F)$$

is effectively decidable

modulo an oracle for deciding $\Rightarrow$

# Relative complete verification

**Ordinary Programs**

$F \in$ FO-Arithmetic

implies

$wp[\![P]\!](F) \in$ FO-Arithmetic

$G \implies wp[\![P]\!](F)$

is effectively decidable

modulo an oracle for deciding $\Rightarrow$

**Probabilistic Programs**

$f \in$ SomeSyntax

implies

$wp[\![P]\!](f) \in$ SomeSyntax

$g \sqsubseteq wp[\![P]\!](f)$

is effectively decidable

modulo an oracle for deciding $\sqsubseteq$

between two syntactic expectations.

# Relative complete verification

## Ordinary Programs

$F \in$ FO-Arithmetic

implies

$wp[\![P]\!](F) \in$ FO-Arithmetic

$G \Longrightarrow wp[\![P]\!](F)$

is effectively decidable

modulo an oracle for deciding $\Rightarrow$

## Probabilistic Programs

$f \in$ SomeSyntax

implies

$wp[\![P]\!](f) \in$ SomeSyntax

$g \sqsubseteq wp[\![P]\!](f)$

is effectively decidable

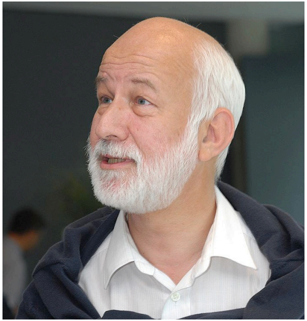
modulo an oracle for deciding $\sqsubseteq$

between two syntactic expectations.

Q: How does the SomeSyntax look like?

# 50 years of Hoare logic

"Completeness is a subtle manner and requires a careful analysis"



Krzysztof R. Apt


Ernst-Rüdiger Olderog

# A syntax for expectations

- Expectations

$$
\begin{array}{llll}
f & \longrightarrow & a & \text{arithmetic expressions} \\
 & | & [\varphi] \cdot f & \text{guarding} \\
 & | & f + f & \text{addition} \\
 & | & a \cdot f & \text{scaling by arithmetic expressions} \\
 & | & \mathcal{S} x \colon f & \text{supremum over variable } x \\
 & | & \mathcal{J} x \colon f & \text{infimum over variable } x
\end{array}
$$

# A syntax for expectations

- Expectations

| | | |
|---|---|---|
| $f \longrightarrow a$ | | arithmetic expressions |
| $\mid [\varphi] \cdot f$ | | guarding |
| $\mid f + f$ | | addition |
| $\mid a \cdot f$ | | scaling by arithmetic expressions |
| $\mid \reflectbox{S}\, x \colon f$ | | supremum over variable $x$ |
| $\mid \reflectbox{L}\, x \colon f$ | | infimum over variable $x$ |

$f \cdot g$

- Examples:

$$\reflectbox{S}\, x \colon [x \cdot x < y] \cdot x \;\equiv\; \sqrt{y} \qquad\qquad \reflectbox{S}\, z \colon [z \cdot (x+1) = 1] \cdot z \;\equiv\; \frac{1}{x+1}$$

$1 \cdot x$

# A syntax for expectations

- Expectations

$$
\begin{array}{lll}
f \longrightarrow & a & \text{arithmetic expressions} \\
& \mid\ [\varphi] \cdot f & \text{guarding} \\
& \mid\ f + f & \text{addition} \\
& \mid\ a \cdot f & \text{scaling by arithmetic expressions} \\
& \mid\ \mathsf{S}\, x \colon f & \text{supremum over variable } x \\
& \mid\ \mathsf{J}\, x \colon f & \text{infimum over variable } x
\end{array}
\Bigg\} \; \text{Exp}
$$

- Examples:

$$\mathsf{S}\, x \colon [x \cdot x < y] \cdot x \;\equiv\; \sqrt{y} \qquad \mathsf{S}\, z \colon [z \cdot (x+1) = 1] \cdot z \;\equiv\; \frac{1}{x+1}$$

- $f \in \mathbb{E}$ is syntactic, if $f$ is expressible in this syntax, i.e., if $f \in \text{Exp}$

# Examples

- polynomials $y + x^3 + 2x^2 + x - 7$ widely used as templates
- rational functions $\dfrac{x^2 - 3x + 4}{y^2 \cdot x - 3y + 1}$
- square roots $\sqrt{x}$
- Harmonic numbers $H_x = \sum_{k=1}^{x} \frac{1}{k}$ used in run-time/termination analysis

# Expressiveness theorem

**[Batz, K. et al., POPL 2021]**

For every pGCL program $P$ and expectation $f \in \text{Exp}$:

$$wp[\![P]\!]([\![f]\!]) \quad = \quad [\![g]\!]$$

for some syntactic expectation $g \in \text{Exp}$.

# Overview

1. Motivation
2. Verifying probabilistic programs
3. Proof rules
4. A syntax for weakest expectations
5. Automation
   - termination
   - verifying invariants
   - synthesising invariants

# The Amber tool

**[Moosbrugger, Kovacs, K., *et al.*, 2021]**

- Simple loops with
  - loop guard $\varphi$: strict inequalities over polynomials
  - loop body: a sequence of random polynomial assignments
- Supports four martingale-based proof rules:
  - PAST, AST, non-AST and non-PAST
- And mild relaxed versions thereof
- Key algorithmic techniques:
  - Algebraic recurrence equations
  - Approximations of polynomial expressions
  - Exact moment-based generation techniques

Automating checking AST and PAST for all inputs

# Program syntax

Programs over $m$ real-valued program variables $x_{(1)}, \ldots, x_{(m)}$:

$$Init\,;\,\texttt{while}(\varphi)\,\{\,P\,\}$$

where:

- $Init$: a sequence of $m$ (random) assignments $x_{(i)} := r_{(i)}$ with $r_{(i)} \in \mathbb{R}$
- $\varphi$: a strict inequality $X > Y$ with $X, Y \in \mathbb{R}[x_{(1)}, \ldots, x_{(m)}]$
- Loop body $P$: a sequence of $m$ probabilistic assignments of the form:

$$x_{(i)} := \text{probabilistic choice over terms of the form } a_{(ij)} \cdot x_{(i)} + X_{ij}$$

where $X_{ij} \in \mathbb{R}[\underbrace{x_{(1)}, \ldots, x_{(i-1)}}_{\text{vars preceding } x_{(i)} \text{ in } P}]$ and $a_{(ij)} \in \mathbb{R}$ are constants

# Experiments: proving PAST

Amber

| Program | | Absynth | MGen | LexRSM | KoAT2 | ecoimp |
|---|---|---|---|---|---|---|
| 2d_bounded_random_walk | ✓ | ✗ | NA | NA | ✗ | ✗ |
| biased_random_walk_const | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| biased_random_walk_exp | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ |
| biased_random_walk_poly | ✓ | ✗ | ✗ | NA | ✗ | ✗ |
| binomial_past | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| complex_past | ✓ | ✗ | NA | NA | ✗ | ✗ |
| consecutive_bernoulli_trails | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| coupon_collector_4 | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ |
| coupon_collector_5 | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ |
| dueling_cowboys | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| exponential_past_1 | ✓ | NA | NA | NA | ✗ | NA |
| exponential_past_2 | ✓ | NA | NA | NA | ✗ | NA |
| geometric | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| geometric_exp | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| linear_past_1 | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ |
| linear_past_2 | ✓ | ✗ | NA | ✗ | ✗ | ✗ |
| nested_loops | NA | ✓ | ✗ | ✓ | ✓ | ✓ |
| polynomial_past_1 | ✓ | ✗ | NA | NA | ✗ | ✗ |
| polynomial_past_2 | ✓ | ✗ | NA | NA | ✗ | ✗ |
| sequential_loops | NA | ✓ | ✗ | ✓ | ✓ | ✓ |
| tortoise_hare_race | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| dependent_dist* | NA | NA | NA | NA | ✗ | ✓ |
| exp_rw_gauss_noise* | ✓ | NA | NA | NA | NA | NA |
| gemoetric_gaussian* | ✓ | NA | NA | NA | NA | NA |
| race_uniform_noise* | ✓ | ✗ | ✓ | ✓ | ✗ | ✓ |
| symb_2d_rw* | ✓ | ✗ | NA | NA | ✗ | ✗ |
| uniform_rw_walk* | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Total ✓ | 23 | 9 | 11 | 12 | 11 | 13 |

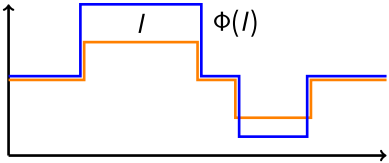
`https://github.com/probing-lab/amber`

# Computing invariants: $k$-induction

Recall Park induction: $\Phi_f(I) \sqsubseteq I$ implies $\underbrace{wp[\![\text{while}(\varphi)\{\text{body}\}]\!](f)}_{= \text{lfp}\, \Phi_f} \sqsubseteq I$

But:

$\text{lfp}\, \Phi_f \sqsubseteq I$ does not imply $\Phi_f(I) \sqsubseteq I$

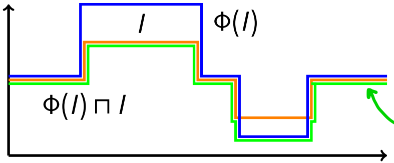


# Computing invariants: *k*-induction

Recall Park induction: $\Phi_f(I) \sqsubseteq I$ implies $\underbrace{wp[\![\text{while}(\varphi)\{\text{body}\}]\!](f)}_{=\ \text{lfp}\ \Phi_f} \sqsubseteq I$

But:
$\text{lfp}\ \Phi_f \sqsubseteq I$ does not imply $\Phi_f(I) \not\sqsubseteq I$

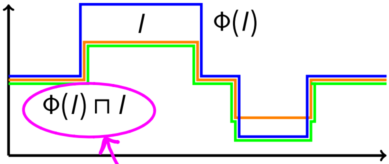
Pointwise minimum: $g \sqcap g' \equiv \lambda s.\ \min\{g(s), g'(s)\}$



# Computing invariants: $k$-induction

Recall Park induction: $\Phi_f(I) \sqsubseteq I$ implies $\underbrace{wp[\![\text{while}(\varphi)\{\text{body}\}]\!](f)}_{= \text{lfp}\, \Phi_f} \sqsubseteq I$

But:
$\text{lfp}\, \Phi_f \sqsubseteq I$ does not imply $\Phi_f(I) \not\sqsubseteq I$



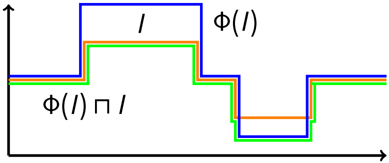

Pointwise minimum: $g \sqcap g' \equiv \lambda s.\, \min\{g(s), g'(s)\}$

2-induction:
$\Phi(\Phi(I) \sqcap I) \sqsubseteq I$ implies $\text{lfp}\ \Phi \sqsubseteq I$

# Computing invariants: *k*-induction

Recall Park induction: $\Phi_f(I) \sqsubseteq I$ implies $\underbrace{wp[\![\text{while}(\varphi)\{\text{body}\}]\!](f)}_{= \text{lfp}\, \Phi_f} \sqsubseteq I$

But:
$\text{lfp}\, \Phi_f \sqsubseteq I$ does not imply $\Phi_f(I) \not\sqsubseteq I$




Pointwise minimum: $g \sqcap g' \equiv \lambda s.\ \min\{g(s), g'(s)\}$

2-induction:
$\Phi(\Phi(I) \sqcap I) \sqsubseteq I$ implies $\text{lfp}\ \Phi \sqsubseteq I$

3-induction:
$\Phi(\Phi(\Phi(I) \sqcap I) \sqcap I) \sqsubseteq I$ implies $\text{lfp}\ \Phi \sqsubseteq I$

# $k$-Induction for probabilistic loops

For a loop while($\varphi$){body} and expectations $f, g, h$, let

$$\Phi_f(g) \;=\; [\varphi] \cdot wp[\![\text{body}]\!](g) + [\neg\varphi] \cdot f \quad \text{and} \quad \Psi_g(h) \;=\; \Phi_f(h) \sqcap g$$

Expectation $I$ is a $k$-inductive invariant if $\Phi_f(\Psi_I^{k-1}(I)) \sqsubseteq I$

$\forall k > 0$, if $I$ is a $k$-inductive invariant, then

$$wp[\![\text{while}(\varphi)\{\text{body}\}]\!](f) \sqsubseteq I$$

# Example

```
pre: s + 1  ✓
post: s

                          ✓ (1-induction)
while (c = 1)  inv s + [c = 1]
{
  { c := 0 } [1/2] { s := s + 1 }
}
```

```
pre: s + 1 ✓
post: s

                          ✓ (2-induction)
while (c = 1)  inv s + 1
{
  { c := 0 } [1/2] { s := s + 1 }
}
```

Tool: https://github.com/moves-rwth/kipro2

# Verifying discrete samplers

```
v := 1; c := 0; term := 0;
while (term = 0) {
  v := 2 · v;
  { c := 2 · c } [1/2] { c := 2 · c + 1 };
  if (v ≥ n) {
    if (c < n) {
      term := 1
    } else {
      v := v − n; c := c − n
    }
  }
}
```

**Optimal Discrete Uniform Generation from Coin Flips, and Applications**

Jérémie Lumbroso

April 9, 2013

For $n \in \{2, 3, 4, 5\}$, we automatically prove

$$\Pr(\text{“sample fixed element K”}) = \text{wp}[\![C]\!]([c = K]) \leqslant 1/n$$

for all $K \in \{0, \ldots, n-1\}$

using 2-, 3-, and 5-induction.

# Inductive invariant synthesis

$fail := 0; sent := 0;$
while $(sent < 8\,000\,000 \wedge fail < 10)$ { fail:=0
  $\{ \underbrace{fail := fail + 1}_{\text{failed transmission}} \}$ [0.01] $\{ \underbrace{sent := sent + 1}_{\text{successful transmission}} \}$
}

**Question:**

- Is the probability of failing to transmit at most 0.05?
- $\mathrm{wp}[\![BRP]\!]([fail = 10]) \leq 0.05$?

**Answer:** ✓

We can prove this using the superinvariant

$$I = \left[\ldots \wedge \frac{13067990199}{280132671650} \cdot fail \leq \frac{5278689867}{211205306866000}\right] \cdot \left(\frac{19 \cdot 8000000 - 19 \cdot sent}{3820000040} + \ldots\right)$$
$$+ \ (7 \text{ more summands})$$

... which fortunately has been synthesized and checked fully automatically.

# Synthesising inductive invariants

Problem: find a piece-wise linear inductive invariant $I$ s.t.

$\underbrace{\Phi_f(I) \sqsubseteq I \text{ and } I \sqsubseteq g}_{I \text{ is inductive for } f \text{ and } g}$ or determine there is no such $I$

# Synthesising inductive invariants

Problem: find a piece-wise linear inductive invariant $I$ s.t.

$$\underbrace{\Phi_f(I) \sqsubseteq I \text{ and } I \sqsubseteq g}_{I \text{ is inductive for } f \text{ and } g} \qquad \text{or determine there is no such } I$$

Approach: use template-based invariants of the (simplified) form:

$$T \;=\; [b_1] \cdot a_1 + \cdots + [b_k] \cdot a_k$$

with

- $b_i$ is a boolean combination of linear inequalities over program vars
- $a_i$ a linear expression over the program variables with $[b_i] \cdot a_i \geq 0$
- the $b_i$'s partition the state space

# Synthesising inductive invariants

Problem: find a piece-wise linear inductive invariant $I$ s.t.

$\underbrace{\Phi_f(I) \sqsubseteq I \text{ and } I \sqsubseteq g}_{I \text{ is inductive for } f \text{ and } g}$ or determine there is no such $I$

Approach: use template-based invariants of the (simplified) form:

$$T \;=\; [b_1] \cdot a_1 + \cdots + [b_k] \cdot a_k$$

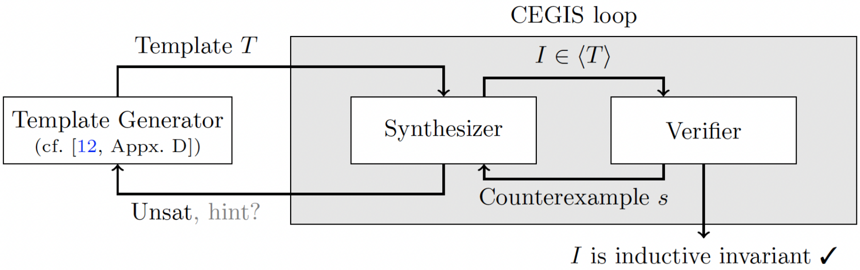
with

- $b_i$ is a boolean combination of linear inequalities over program vars
- $a_i$ a linear expression over the program variables with $[b_i] \cdot a_i \geq 0$
- the $b_i$'s partition the state space

Example: $[c{=}1] \cdot (2{\cdot}x + 1) + [c{\neq}1] \cdot x$ is in the above form,
and $[x \geq 1] \cdot x + [x \geq 2] \cdot y$ can be rewritten into it.

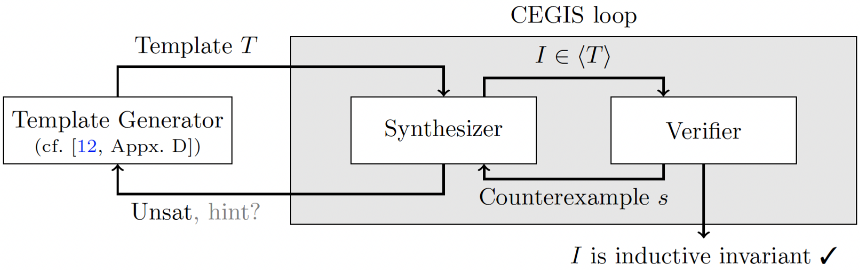
# CEGIS for probabilistic invariants

**[Batz, K. et al., TACAS 2023]**



# CEGIS for probabilistic invariants

**[Batz, K. et al., TACAS 2023]**

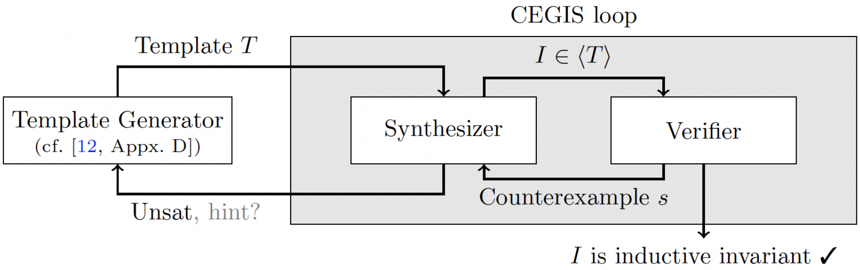



- For finite-state programs, synthesis is sound and complete
- Applicable to lower bounds: UPAST and difference boundedness

- Uses SMT with QF-LRA (the synthesiser) and QF-LIRA (the verifier)

# CEGIS for probabilistic invariants

[Batz, K. et al., TACAS 2023]

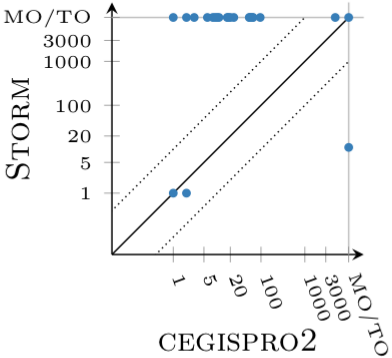



- For finite-state programs, synthesis is sound and complete
- Applicable to lower bounds: UPAST and difference boundedness

- Uses SMT with QF-LRA (the synthesiser) and QF-LIRA (the verifier)

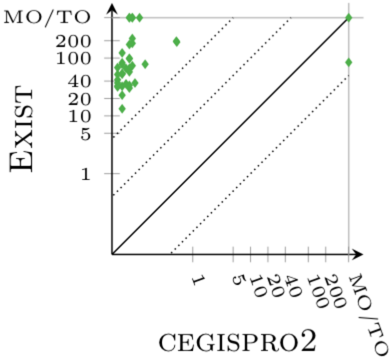
CEGISPRO2 tool: `https://github.com/moves-rwth/cegispro2`

# Experiments




Synthesis of upper bounds for finite-state programs

TO = 2h, MO = 8GB




Synthesis of lower bounds

TO = 5min

# Epilogue

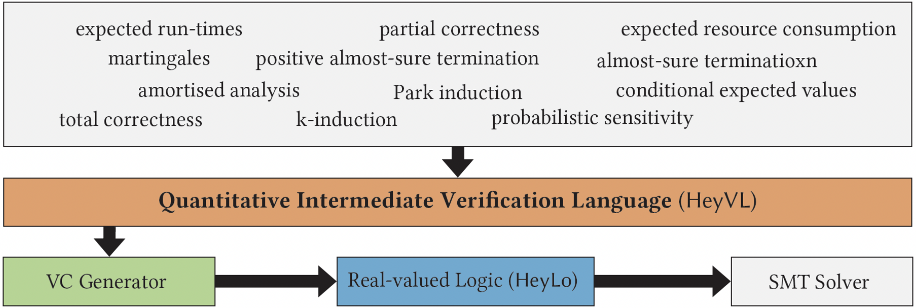
- Weakest preconditions nicely fit analysis of probabilistic programs
- Several extensions of Kozen's seminal work have been developed

  expected run-times, recursion, separation logic, semi-rings, etc.
- And have been equipped with powerful proof rules

  lower bounds, upper bounds, (non-)AST, (non-)PAST . . .
- A syntax to express quantitative measures
- Promising results towards automated analysis of loops (and recursion)

# Outlook: probabilistic Viper/Dafny?

WIP




A verification infrastructure for probabilistic programs

https://caesarverifier.org

# A big thanks to my co-workers!

Ezio Bartocci | Kevin Batz | Mingshuai Chen | Sebastian Junges | Benjamin Kaminski | Laura Kovacs | Lutz Klinkenberg

Christoph Matheja | Annabelle McIver | Marcel Moosbrugger | Carroll Morgan | Federico Olmedo | Philipp Schroer | Tobias Winkler