Probabilistic Programming
Fun but Intricate Too!

Joost-Pieter Katoen

with Friedrich Gretz, Nils Jansen, Benjamin Kaminski
Christoph Matheja, Federico Olmedo and Annabelle McIver

Mysore Workshop on Quantitative Verification, February 2016
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.”

---

\textsuperscript{aMIT/EECS George M. Sprowls Doctoral Dissertation Award}
A 48M US dollar research program

Probabilistic Programming for Advancing Machine Learning (PPAML)
Probabilistic programs

What are probabilistic programs?
Sequential programs with random assignments and conditioning.

Applications
Security, machine learning, quantum computing, approximate computing

Almost every programming language has a probabilistic variant
Probabilistic C, Figaro, ProbLog, R2, Tabular, Rely, ....
Aim of this work

What do we want to achieve?
Formal reasoning about probabilistic programs à la Floyd-Hoare.

What do we need?
Rigorous semantics of random assignments and conditioning.

Approach

1. Develop a wp-style semantics with proof rules for loops
2. Show the correspondence to an operational semantics
3. Study the extension with non-determinism
4. Applications: Prove program transformations, program correctness, program equivalence, and expected run-times of programs

We consider an “assembly” language: probabilistic guarded command language
Roadmap of this talk

1. Introduction
2. Two flavours of semantics
3. Program transformations and equivalence
4. Recursion
5. Non-determinism
6. Different flavours of termination
7. Run-time analysis
8. Synthesizing loop invariants
9. Epilogue
Dijkstra’s guarded command language

- skip
- abort
- \( x := E \)
- \( \text{prog1 ; prog2} \)
- \( \text{if (G) prog1 else prog2} \)
- \( \text{prog1 [] prog2} \)
- \( \text{while (G) prog} \)
Conditional probabilistic GCL  \( \text{cpGCL} \)

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

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

[Hicks 2014, The Programming Languages Enthusiast]

“The crux of probabilistic programming is to consider normal-looking programs as if they were probability distributions.”
A loopy program

For $p$ an arbitrary probability:

```plaintext
bool c := true;
int i := 0;
while (c) {
    i := i + 1;
    (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 := i + 1;
    (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

\[
\begin{align*}
x &:= 0 \ [0.5] \ x := 1; \\
y &:= -1 \ [0.5] \ y := 0; \\
\textit{observe} \ (x+y = 0)
\end{align*}
\]

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 $p$ an arbitrary probability:

```c
bool c := true;
int i := 0;
while (c) {
    i := i + 1;
    (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 models the following distribution with parameter $p$:

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

$$Pr[i = 2N] = 0$$
Operational semantics

This can be defined using Plotkin’s SOS-style semantics
The piranha problem

One fish is contained within the confines of an opaque fishbowl. The fish is equally likely to be a piranha or a goldfish. A sushi lover throws a piranha into the fish bowl alongside the other fish. Then, immediately, before either fish can devour the other, one of the fish is blindly removed from the fishbowl. The fish that has been removed from the bowl turns out to be a piranha. What is the probability that the fish that was originally in the bowl by itself was a piranha?
Operational semantics

\[
f_1 := g f [0.5] f_1 := \text{pir}; \\
f_2 := \text{pir}; \\
s := f_1 [0.5] s := f_2; \\
\text{observe } (s = \text{pir})
\]

What is the probability that the original fish in the bowl was a piranha?

Consider the expected reward of successful termination without violating any observation

\[
\text{cer}(P, [f_1 = \text{pir}]) = \frac{1 \cdot \frac{1}{2} + 0 \cdot \frac{1}{4}}{1 - \frac{1}{4}} = \frac{\frac{1}{2}}{\frac{3}{4}} = \frac{2}{3}.
\]
# Expectations

<table>
<thead>
<tr>
<th>Weakest pre-expectation</th>
<th>[McIver &amp; Morgan 2004]</th>
</tr>
</thead>
<tbody>
<tr>
<td>An expectation maps program states onto non-negative reals. It’s the quantitative analogue of a predicate.</td>
<td></td>
</tr>
<tr>
<td>An expectation transformer is a total function between two expectations on the state of a program.</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>
<tr>
<td>Annotation ${e} P {f}$ holds for total correctness iff $e \leq wp(P, f)$, where $\leq$ is to be interpreted in a point-wise manner.</td>
<td></td>
</tr>
</tbody>
</table>

Weakest liberal pre-expectation $wl_p(P, f) = wp(P, f) + Pr[P$ diverges].
## Expectation transformer semantics of \( \text{cpGCL} \)

### 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.

\( wp \)-semantics differs from \( wp \)-semantics only for `while` and `abort`. 
Probabilistic Programming is Fun, but Intricate Too

Two flavours of semantics

\[
x := 0 \quad [1/2] \quad x := 1; \quad \quad // \quad \text{command } c1
\]

\[
y := 0 \quad [1/3] \quad y := 1; \quad \quad // \quad \text{command } c2
\]

\[
wp(c_1; c_2, [x = y])
\]

\[
= wp(c_1, wp(c_2, [x = y]))
\]

\[
= wp(c_1, \frac{1}{3} \cdot wp(y := 0, [x = y]) + \frac{2}{3} \cdot wp(y := 1, [x = y]))
\]

\[
= wp(c_1, \frac{1}{3} \cdot [x = 0] + \frac{2}{3} \cdot [x = 1])
\]

\[
= \frac{1}{2} \cdot wp(x := 0, \frac{1}{3} \cdot [x = 0] + \frac{2}{3} \cdot [x = 1]) + \frac{1}{2} \cdot wp(x := 1, \frac{1}{3} \cdot [x = 0] + \frac{2}{3} \cdot [x = 1])
\]

\[
= \frac{1}{2} \cdot (\frac{1}{3} \cdot [0 = 0] + \frac{2}{3} \cdot [0 = 1]) + \frac{1}{2} \cdot (\frac{1}{3} \cdot [1 = 0] + \frac{2}{3} \cdot [1 = 1])
\]

\[
= \frac{1}{2} \cdot (\frac{1}{3} \cdot 1 + \frac{2}{3} \cdot 0) + \frac{1}{2} \cdot (\frac{1}{3} \cdot 0 + \frac{2}{3} \cdot 1)
\]

\[
= \frac{1}{2} \cdot (\frac{1}{3} + \frac{2}{3})
\]

\[
= \frac{1}{2}
\]
The piranha program – a wp perspective

\[
\begin{align*}
\text{f1 := gf [0.5] f1 := pir; } \\
\text{f2 := pir; } \\
\text{s := f1 [0.5] s := f2; } \\
\text{observe (s = pir)}
\end{align*}
\]

What is the probability that the original fish in the bowl was a piranha?

\[
\mathbb{E}(f1 = \text{pir} \mid P \text{ terminates}) = \frac{1 \cdot 1/2 + 0 \cdot 1/4}{1 - 1/4} = \frac{1/2}{3/4} = \frac{2}{3}.
\]

We define \( cwp(P, f) = \frac{wp(P, f)}{wlp(P, 1)} \).

\( wlp(P, 1) = 1 - Pr[P \text{ violates an observation}] \). This includes diverging runs.
Divergence matters

```
abort [0.5] {
    x := 0 [0.5] x := 1;
    y := 0 [0.5] y := 1;
    observe (x = 0 || y = 0)
}
```

Our approach: \( \frac{wp(P, f)}{wlp(P, 1)} \)

Here: \( cwp(P, [y = 0]) = \frac{2}{7} \)

Microsoft's R2 approach: \( \frac{wp(P, f)}{wp(P, 1)} \)

Here: \( cwp(P, [y = 0]) = \frac{2}{3} \)

In general:

\( \text{observe} (G) \equiv \text{while}(!G) \text{ skip} \)

**Warning:** This is a simple example. Typically divergence comes from loops.
Leave divergence up to the programmer?

Almost-sure termination is “more undecidable” than ordinary termination. More on this follows later.
Infeasible programs

\[
\begin{align*}
\text{int } & \ x := 1; \\
\text{while } & \ (x = 1) \{ \\
\quad & \ x := 1 \\
\} \\
\end{align*}
\]

▶ Certain divergence

\[
\begin{align*}
\text{int } & \ x := 1; \\
\text{while } & \ (x = 1) \{ \\
\quad & \ x := 1 \ [0.5] \ x := 0; \\
\quad & \text{observe } (x = 1) \\
\} \\
\end{align*}
\]

▶ Divergence with probability zero.

▶ Conditional termination = undefined.

These two programs are mostly not distinguished. We do.
Soundness?

Our wp-semantics is a conservative extension of McIver’s wp-semantics.

McIver’s wp-semantics is a conservative extension of Dijkstra’s wp-semantics.
Weakest pre-expectations = conditional rewards

For program $P$ and expectation $f$ with $cwp(P, f) = (wp(P, f), wlp(P, 1))$:

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

---

\(^1\)Either both sides are equal or both sides are undefined.
Overview

1 Introduction
2 Two flavours of semantics
3 Program transformations and equivalence
4 Recursion
5 Non-determinism
6 Different flavours of termination
7 Run-time analysis
8 Synthesizing loop invariants
9 Epilogue
Importance of these results

- Unambiguous meaning to (almost) all probabilistic programs
- Operational interpretation to weakest pre-expectations
- Basis for proving correctness
  - of programs
  - of program transformations
  - of program equivalence
  - of static analysis
  - of compilers
  - ........
Removal of conditioning

- **Idea:** restart an infeasible run until all observe-statements are passed

- **Change** `prog` by adding auxiliary variable `flag` and:
  - `observe(G)` becomes `if(~G)flag := true`
  - `abort` becomes `if(~flag)abort`
  - `while(G) prog` becomes `while(G && ~flag)prog`

- **For program variable** `x` use auxiliary variable `sx`
  - store initial value of `x` into `sx`
  - on each new loop-iteration restore `x` to `sx`

```plaintext
sx1,...,sxn := x1,...,xn; flag := true;
while(flag) {
    flag := false;
    x1,...,xn := sx1,...,sxn;
    modprog
}
```

where modprog is obtained from prog as above
Removal of conditioning

the transformation in action:

\[
\begin{align*}
\text{x := 0} & \quad \text{[p]} \quad \text{x := 1;}
\text{y := 0} & \quad \text{[p]} \quad \text{y := 1;}
\text{observe}(x /= y)
\end{align*}
\]

\[
\text{sx, sy := x, y; flag := true;}
\text{while}(\text{flag}) \{
\text{x, y := sx, sy; flag := false;}
\text{x := 0} & \quad \text{[p]} \quad \text{x := 1;}
\text{y := 0} & \quad \text{[p]} \quad \text{y := 1;}
\text{if (x = y) flag := true}
\}
\]

a data-flow analysis yields:

\[
\begin{align*}
\text{x, y := 0, 0;}
\text{while}(x /= y) \{
\text{x := 0} & \quad \text{[p]} \quad \text{x := 1;}
\text{y := 0} & \quad \text{[p]} \quad \text{y := 1}
\}
\]
Removal of conditioning

Soundness of transformation

For program $P$, transformed program $\hat{P}$, and post-expectation $f$:

$$cwp(P, f) = wp(\hat{P}, f)$$
A dual program transformation

\[
\begin{align*}
\text{repeat} & \\
& a_0 := 0 \ [0.5] \ a_0 := 1; \\
& a_1 := 0 \ [0.5] \ a_1 := 1; \\
& a_2 := 0 \ [0.5] \ a_2 := 1; \\
& i := 4a_0 + 2a_1 + a_0 + 1 \\
\text{until } (1 \leq i \leq 6)
\end{align*}
\]

\[
\begin{align*}
a_0 := 0 \ [0.5] \ a_0 := 1; \\
a_1 := 0 \ [0.5] \ a_1 := 1; \\
a_2 := 0 \ [0.5] \ a_2 := 1; \\
i := 4a_0 + 2a_1 + a_0 + 1 \\
\text{observe } (1 \leq i \leq 6)
\end{align*}
\]

Loop-by-observe replacement if there is no data flow between loop iterations
Playing with geometric distributions

- $X$ is a random variable, geometrically distributed with parameter $p$
- $Y$ is a random variable, geometrically distributed with parameter $q$

Q: generate a sample $x$, say, according to the random variable $X - Y$

```c
int XmInY1(float p, q){ // 0 <= p, q <= 1
    int x := 0;
    bool flip := false;
    while (not flip) { // take a sample of X to increase x
        (x += 1 [p] flip := true);
    }
    flip := false;
    while (not flip) { // take a sample of Y to decrease x
        (x -= 1 [q] flip := true);
    }
    return x; // a sample of X-Y
}
```
Program equivalence

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

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

Claim: [Kiefer et al., 2012]
Both programs are equivalent for (p, q) = (1/2, 2/3).

Our (semi-automated) analysis yields:
Both programs are equivalent for any q with \( q = \frac{1}{2-p} \).
Overview

1. Introduction
2. Two flavours of semantics
3. Program transformations and equivalence
4. Recursion
5. Non-determinism
6. Different flavours of termination
7. Run-time analysis
8. Synthesizing loop invariants
9. Epilogue
Recursion

Can we also deal with recursion, such as:

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

For instance, with which probability does \( P \) terminate?
Recursion

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

\[
\begin{align*}
\text{call}^D_0 P & = \text{abort} \\
\text{call}^D_{n+1} P & = D(P)[\text{call} P := \text{call}^D_n P]
\end{align*}
\]

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

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 \( f \):

\[ w_P(P, f) \text{ for input } \eta \text{ equals the expected reward (that depends on } f \text{) to reach a terminal state in the pushdown MC of } P \text{ when starting with } \eta. \]

Checking expected rewards in finite-control pushdown MDPs is decidable.
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 \( P' \) 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 \( u_0 = 0 \):

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

\[ \sup_n \ell_n \leq wp(\text{call } P, f)[D] \leq \sup_n u_n \]
The golden ratio

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

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

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.
\[
w_{\text{call } P}(1) \leq \varphi \quad \vdash \quad w_{\mathcal{D}(P_{\text{rec3}})}(1) \leq \varphi
\]

\[
w_{\mathcal{D}(P_{\text{rec3}})}(1)
\begin{align*}
= & \quad \text{(def. of \(w_p\))} \\
= & \quad \frac{1}{2} \cdot w_{\text{skip}}(1) + \frac{1}{2} \cdot w_{\text{call } P_{\text{rec3}}; \text{ call } P_{\text{rec3}}; \text{ call } P_{\text{rec3}}}(1) \\
\quad \text{(def. of \(w_p\))} \\
= & \quad \frac{1}{2} + \frac{1}{2} \cdot w_{\text{call } P_{\text{rec3}}; \text{ call } P_{\text{rec3}}}(w_{\text{call } P_{\text{rec3}}}(1)) \\
\quad \text{(assumption, monoton. of \(w_p\))} \\
\geq & \quad \frac{1}{2} + \frac{1}{2} \cdot w_{\text{call } P_{\text{rec3}}; \text{ call } P_{\text{rec3}}}(\varphi) \\
\quad \text{(def. of \(w_p\), scalab. of \(w_p\) twice)} \\
\geq & \quad \frac{1}{2} + \frac{1}{2} \varphi \cdot w_{\text{call } P_{\text{rec3}}}(w_{\text{call } P_{\text{rec3}}}(1)) \\
\quad \text{(assumption, monoton. of \(w_p\))} \\
\geq & \quad \frac{1}{2} + \frac{1}{2} \varphi \cdot w_{\text{call } P_{\text{rec3}}}(\varphi) \\
\quad \text{(scalab. of \(w_p\))} \\
\geq & \quad \frac{1}{2} + \frac{1}{2} \varphi^2 \cdot w_{\text{call } P_{\text{rec3}}}(1) \\
\quad \text{(assumption, monoton. of \(w_p\))} \\
\geq & \quad \frac{1}{2} + \frac{1}{2} \varphi^3 \\
\quad \text{(algebra)} \\
\geq & \quad \varphi
\]

\[\triangle\]
Overview

1. Introduction
2. Two flavours of semantics
3. Program transformations and equivalence
4. Recursion
5. Non-determinism
6. Different flavours of termination
7. Run-time analysis
8. Synthesizing loop invariants
9. Epilogue
Non-determinism

"[...] there are several technical challenges in adding non-determinism to probabilistic programs."

[Gordon, Henzinger et al. 2014]
Non-determinism: Operational semantics

- Use Markov decision processes (rather than Markov chains)
- Resolve the non-determinism by means of policies
- Take expected rewards over demonic policies:

\[
C\text{ExpRew}^{\mathcal{R}}(\Diamond T \mid \neg \Diamond U) \triangleq \inf_{\mathcal{S} \in Sched^{\mathcal{R}}} \frac{\text{ExpRew}^{\mathcal{R}}(\Diamond T \cap \neg \Diamond U)}{\text{Pr}^{\mathcal{R}}(\neg \Diamond U)}
\]

Simple extension. But: conditioning needs policies with memory.
Non-determinism

\[
\begin{align*}
\text{x := 1} & \ [1/2] \ \{ \text{x := 2} \ [\] \ \{ \text{observe(false)} \ [1/2] \text{ x := 2.2}\} \} \\
\end{align*}
\]

Cond. Exp. starting in \( s_2 \) = \( \frac{2}{1} = 2 \) (taking action \( \mu \)).

Cond. Exp. starting in \( s_0 \) = \( \frac{1/2 \cdot 1 + 1/4 \cdot 2.2}{3/4} = 1.46 \) (taking action \( \nu \)).
Non-determinism: \( wp \)-semantics

Without conditioning:

\[
wp(P_1 \parallel P_2, f) = \min(wp(P_1, f), wp(P_2, f))
\]

This corresponds to a demonic resolution of non-determinism

This preserves the correspondence to the operational semantics
Non-determinism + conditioning is problematic

The non-deterministic choice \( \{P_1\} \circ \{P_2\} \) is an implementation choice. More formally: If it holds that

\[
cwp[\{P_1\} \circ \{P_2\}] = cwp[P_1]
\]

then it should also hold that

\[
cwp[\{\{P_1\} \circ \{P_2\}\} \{p\} \{P_3\}] = cwp[\{P_1\} \{p\} \{P_3\}].
\]

It is **impossible** to provide a compositional wp-semantics for non-determinism in presence of conditioning.\(^2\)

\(^2\)Under the assumption that non-determinism is an implementation choice.
Probabilistic Programming is Fun, but Intricate Too

Non-determinism

Probabilistic Programming

Joost-Pieter Katoen

1/2

1/2

P : \( \{x := 0\} [1/2] \{x := 1\}; \text{observe}(x = 1) \)

Q : \( \{x := 0; \text{observe}(x = 1)\} [1/2] \{x := 1; \text{observe}(x = 1)\} \)

Of course

\[
\frac{wp(P, [x = 1])}{wlp(P, 1)} = \frac{wp(Q, [x = 1])}{wlp(Q, 1)} = \frac{1/2}{1/2} = 1
\]
Probabilistic Programming is Fun, but Intricate Too

Non-determinism

\[
P : \quad \{ x := 0 \} [1/2] \{ x := 1 \}; \ \text{observe}(x = 1)
\]

\[
Q : \quad \{ x := 0; \ \text{observe}(x = 1) \} [1/2] \{ x := 1; \ \text{observe}(x = 1) \}
\]

Of course

\[
\frac{wp(P, [x = 1])}{wp(P, 1)} = \frac{wp(Q, [x = 1])}{wp(Q, 1)} = \frac{1/2}{1/2} = 1
\]

but we cannot decompose

\[
\frac{wp(Q, [x = 1])}{wp(Q, 1)} \neq 0.5 \frac{wp(Q_1, [x = 1])}{wp(Q_1, 1)} + 0.5 \frac{wp(Q_2, [x = 1])}{wp(Q_2, 1)}
\]
Overview

1. Introduction
2. Two flavours of semantics
3. Program transformations and equivalence
4. Recursion
5. Non-determinism
6. Different flavours of termination
7. Run-time analysis
8. Synthesizing loop invariants
9. Epilogue
“[Ordinary] termination is a purely topological property [...], but almost-sure termination is not. [...] Proving almost–sure termination requires arithmetic reasoning not offered by termination provers.”
Nuances of termination

...... certain termination

...... termination with probability one

⇒ almost-sure termination

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

⇒ positive almost-sure termination

...... for all possible program inputs

⇒ universal [positive] almost-sure termination
Certain termination

```c
int i := 100;
while (i > 0) {
    i := i - 1;
}
```

This program **certainly** terminates.
Positive almost-sure termination

For $p$ an arbitrary probability:

```plaintext
bool c := true;
int i := 0;
while (c) {
    i := i + 1;
    (c := false [p] c := true)
}
```

This program almost surely terminates. In finite expected time.
Negative almost-sure termination

Consider the one-dimensional (symmetric) random walk:

```c
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:

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

**Finite** expected termination time

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

**Finite** termination time

Running the right after the left program yields an **infinite** expected termination time
Three results

Determining expected outcomes is as hard as almost-sure termination.

Almost-sure termination is “more undecidable” than ordinary termination.

Universal almost-sure termination is as hard as almost-sure termination.
This does not hold for positive almost-sure termination.
Hardness of almost sure termination

\[
\begin{align*}
\Sigma_3^0 & \quad \Delta_3^0 & \quad \Pi_3^0 \\
\Sigma_2^0 & \quad REXP \quad PAST & \quad \Delta_2^0 & \quad AST \quad UAST \quad EXP & \quad \Pi_2^0 \\
\Sigma_1^0 & \quad LEXP & \quad \Delta_1^0 & \quad \Pi_1^0 \\
\mathcal{H} & \quad \overline{\mathcal{H}} \\
\text{semi-decidable} & \quad \text{semi-decidable} & \quad \text{not semi-decidable; even with access to } \mathcal{H}-\text{oracle} & \quad \text{not semi-decidable; even with access to } \mathcal{UH}-\text{oracle}
\end{align*}
\]
Proof idea: hardness of positive as-termination

Reduction from the complement of the universal halting problem

For an ordinary program $Q$ that does not on all inputs terminate, provide a probabilistic program $P$ (depending on $Q$) and an input $\eta$, such that $P$ does terminate in an expected finite number of steps on $\eta$. 
Let’s start simple

```c
bool c := true;
int nrflips := 0;
while (c) {
    nrflips := nrflips + 1;
    (c := false [0.5] c := true);
}
```

Expected runtime (integral over the bars):

The nrflips-th iteration takes place with probability $\frac{1}{2^{nrflips}}$. 
Reducing an ordinary program to a probabilistic one

Assume an enumeration of all inputs for $Q$ is given

```c
bool c := true;
int nrflips := 0;
int i := 0;
while (c) {
    // simulate $Q$ for one (further) step on its $i$-th input
    if (Q terminates on its $i$th input) {
        i := i + 1;
        // reset simulation of program $Q$
        cheer // take $2^{nrflips}$ meaningless steps
    } else {
        nrflips := nrflips + 1;
        (c := false [0.5] c := true);
    }
}
```

$P$ loses interest in further simulating $Q$ by a coin flip to decide for termination.
Q does not always halt

Let $i$ be the first input for which $Q$ does not terminate.

Expected runtime of $P$ (integral over the bars):

Finite **cheering** — finite expected runtime!
Q terminates on all inputs

Expected runtime of $P$ (integral over the bars):

Infinite cheering — infinite expected runtime!
Overview

1. Introduction
2. Two flavours of semantics
3. Program transformations and equivalence
4. Recursion
5. Non-determinism
6. Different flavours of termination
7. Run-time analysis
8. Synthesizing loop invariants
9. Epilogue
Expected run-times

Aim

Provide a wp-calculus to determine expected run-times. Why?

1. Be able to prove positive almost-sure termination
2. Reason about the efficiency of randomised algorithms

Let $ert() : T \rightarrow T$ where $T = \{ t \mid t : S \rightarrow [0, \infty] \}$

$ert(P, t)$ represents the run-time of $P$ given that its continuation takes $t$ time units
## Expected run-times

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Semantics ert(P, t)</th>
</tr>
</thead>
<tbody>
<tr>
<td>▶ skip</td>
<td>▶ 1 + t</td>
</tr>
<tr>
<td>▶ abort</td>
<td>▶ 0</td>
</tr>
<tr>
<td>▶ x := mu</td>
<td>▶ 1 + ( \lambda \sigma. E_{\mu}(\sigma) (\lambda v. t<a href="%5Csigma">x := v</a>) )</td>
</tr>
<tr>
<td>▶ P1 ; P2</td>
<td>▶ ert(P₁, ert(P₂, t))</td>
</tr>
<tr>
<td>▶ if (G) P1 else P2</td>
<td>▶ 1 + [G] · ert(P₁, t) + [¬G] · ert(P₂, t)</td>
</tr>
<tr>
<td>▶ P1 [] P2</td>
<td>▶ max (ert(P₁, t), ert(P₂, t))</td>
</tr>
<tr>
<td>▶ while(G)P</td>
<td>▶ ( \mu X. 1 + ([G] · ert(P, X) + [¬G] · t) )</td>
</tr>
</tbody>
</table>

\( \mu \) is the least fixed point operator wrt. the ordering \( \leq \) on run-times

accompanied with a set of proof rules to get two-sided bounds on run-times
Coupon collector problem

A more modern phrasing:
Each box of cereal contains one (equally likely) out of $N$ coupons. You win a price if all $N$ coupons are collected. How many boxes of cereal need to be bought on average to win?
Coupon collector problem

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

Using our ert-calculus one can prove that expected run-time is \( \Theta(N \cdot \log N) \).
By systematic formal verification à la Floyd-Hoare. No hidden assumptions.
Overview

1. Introduction
2. Two flavours of semantics
3. Program transformations and equivalence
4. Recursion
5. Non-determinism
6. Different flavours of termination
7. Run-time analysis
8. Synthesizing loop invariants
9. Epilogue
Quantitative loop invariants

Recall that for while-loops we have:

\[ wp(\text{while}(G)\{P\}, f) = \mu X. ([G] \cdot wp(P, X) + [\neg G] \cdot f) \]

To determine this \( wp \), we use an “invariant” \( I \) such that \([\neg G] \cdot I \leq f\).

Quantitative loop invariant

Expectation \( I \) is a quantitative loop invariant if —by consecution—

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

To guarantee soundness, \( I \) has to fulfill either:

1. \( I \) is bounded from below and by above by some constants, or
2. on each iteration there is a probability \( \epsilon > 0 \) to exit the loop

Then: \( \{ I \} \) while\((G)\{P\} \{ f \} \) is a correct program annotation.
Invariant synthesis for linear programs

1. Speculatively annotate a while-loop with linear expressions:

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

with real parameters \(\alpha_i, \beta_i\), program variable \(x_i\), and \(\ll \in \{<, \leq\}\).
2. Transform these numerical constraints into Boolean predicates.
3. Transform these predicates into non-linear FO formulas.
4. Use constraint-solvers for quantifier elimination (e.g., REDLOG).
5. Simplify the resulting formulas (e.g., using SLFQ and SMT solving).
6. Exploit resulting assertions to infer program correctness.
Soundness and completeness

For any linear pGCL program annotated with propositionally linear expressions, our method will find all parameter solutions that make the annotation valid, and no others.
**Prinsys Tool: Synthesis of Probabilistic Invariants**

- **Probabilistic Program**
  - User input
  - Parser
  - Probabilistic verification conditions VC
  - Transformation to DNF
  - VC in DNF

- **Template**
  - User input
  - WLP computation
  - Transformation to DNF
  - Translation to FO-formula

- **FO-formula equivalent to VC**
  - REDLOG
  - Quantifier elimination
  - Quantifier-free constraints on template parameters
  - SLFQ
  - User chooses parameter values

- **Invariant**

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

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

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

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

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

Joost-Pieter Katoen
Epilogue

Take-home message

- Connection between wp and operational semantics
- Semantic intricacies of conditioning (divergence)
- Interplay of non-determinism and conditioning
- Program transformations

Extensions

- Recursion
- Loop invariant synthesis
- Expected run-time analysis
- Intricacies of termination
Further reading

- **J.-P. K., A. McIver, L. Meinicke, and C. Morgan.**
  *Linear-invariant generation for probabilistic programs.*
  SAS 2010.

- **F. Gretz, J.-P. K., and A. McIver.**
  *Operational versus wp-semantics for pGCL.*

- **F. Gretz et al..**
  *Conditioning in probabilistic programming.*
  MFPS 2015.

- **B. Kaminski, J.-P. K., C. Matheja, and F. Olmedo**
  *Determining expected run-times of probabilistic programs.*
  ESOP 2016\(^3\).

- **B. Kaminski, J.-P. K., C. Matheja, and F. Olmedo**
  *Reasoning about recursive probabilistic programs.*
  submitted.

\(^3\)Nominated for the EATCS best paper award of ETAPS 2016.