Probabilistic Programming

Fun but Intricate Too!

Joost-Pieter Katoen

Software Modeling and Verification Chair

RWTH AACHEN UNIVERSITY

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

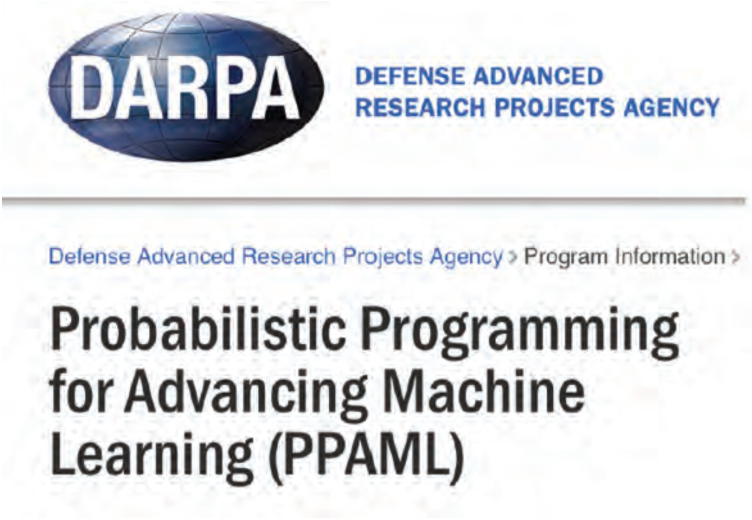
[Daniel Roy, 2011][a]

"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

A 48M US dollar research program



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

Dijkstra's guarded command language

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

Conditional probabilistic GCL `cpGCL`

- `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:

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

[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:

```
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.

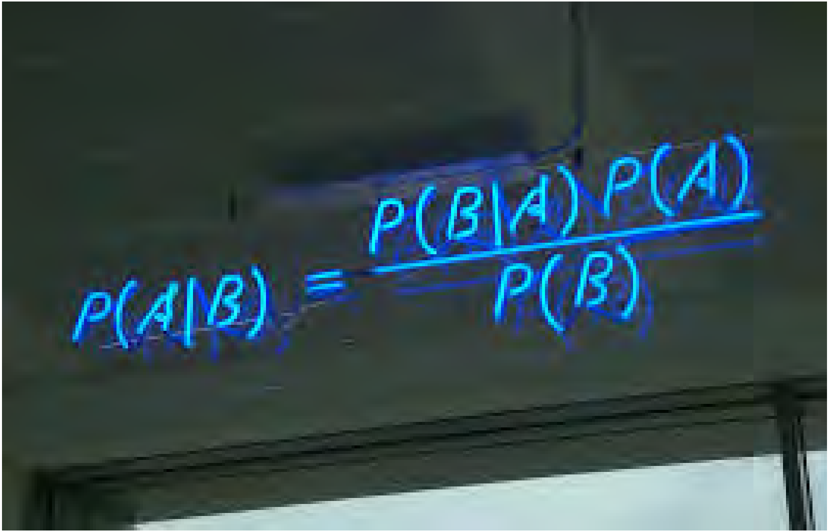
$$Pr[i = N] \;=\; (1-p)^{N-1} \cdot p \quad \text{for } N > 0$$

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



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 p an arbitrary probability:

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

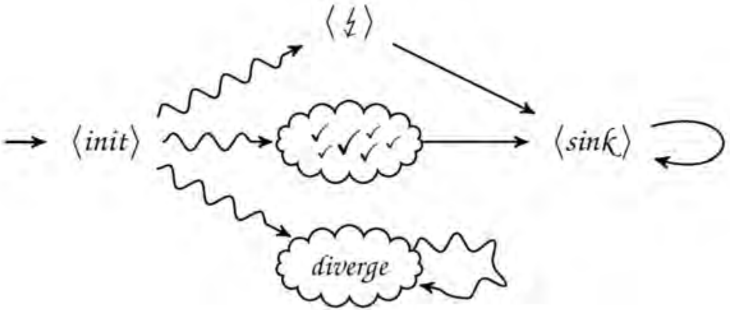
The feasible program runs have a probability $\sum_{N \geqslant 0} (1-p)^{2N} \cdot p \ = \ {}^{1}/_{(2-p)}$

This models the following distribution with parameter p:

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

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

Operational semantics



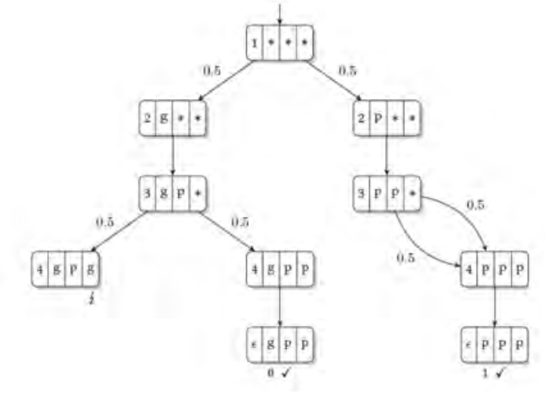
This can be defined using Plotkin's SOS-style semantics

The piranha problem [Tijms, 2004]

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

```
f1 := gf [0.5] f1 := pir;
f2 := pir;
s := f1 [0.5] s := f2;
observe (s = 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

$$\mathrm{cer}(P, f)(\sigma_I) = \frac{\mathrm{ExpRew}^{\mathfrak{R}[\![P]\!]}(\Diamond\, sink \cap \neg\Diamond \lightning)}{\Pr(\neg\Diamond \lightning)}$$

$$\mathrm{cer}(P, [\mathtt{f1\ =\ pir}])(\sigma_I) = \frac{1 \cdot 1/2 + 0 \cdot 1/4}{1 - 1/4} = \frac{1/2}{3/4} = 2/3.$$

Expectations

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

An expectation maps program states onto non-negative reals. It's the quantitative analogue of a predicate.

An expectation transformer is a total function between two expectations on the state of a program.

The transformer $wp(P, f)$ for program P and post-expectation f yields the least 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 $e \leqslant wp(P, f)$, where $\leqslant$ is to be interpreted in a point-wise manner.

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

Expectation transformer semantics of cpGCL

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 $\leqslant$ on expectations.

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

```
x := 0 [1/2] x := 1; // command c1
y := 0 [1/3] y := 1; // command c2
```

$$
\begin{array}{c}
wp(c_1; c_2, [x = y]) \\
= \\
wp(c_1, wp(c_2, [x = y])) \\
= \\
wp(c_1, {}^1\!/_3 \cdot wp(y := 0, [x = y]) + {}^2\!/_3 \cdot wp(y := 1, [x = y])) \\
= \\
wp(c_1, {}^1\!/_3 \cdot [x = 0] + {}^2\!/_3 \cdot [x = 1]) \\
= \\
{}^1\!/_2 \cdot wp(x := 0, {}^1\!/_3 \cdot [x = 0] + {}^2\!/_3 \cdot [x = 1]) + {}^1\!/_2 \cdot wp(x := 1, {}^1\!/_3 \cdot [x = 0] + {}^2\!/_3 \cdot [x = 1]) \\
= \\
{}^1\!/_2 \cdot ({}^1\!/_3 \cdot [0 = 0] + {}^2\!/_3 \cdot [0 = 1]) + {}^1\!/_2 \cdot ({}^1\!/_3 \cdot [1 = 0] + {}^2\!/_3 \cdot [1 = 1]) \\
= \\
{}^1\!/_2 \cdot ({}^1\!/_3 \cdot \mathbf{1} + {}^2\!/_3 \cdot \mathbf{0}) + {}^1\!/_2 \cdot ({}^1\!/_3 \cdot \mathbf{0} + {}^2\!/_3 \cdot \mathbf{1}) \\
= \\
{}^1\!/_2 \cdot ({}^1\!/_3 + {}^2\!/_3) \\
= \\
{}^1\!/_2
\end{array}
$$

The piranha program – a wp perspective

```
f1 := gf [0.5] f1 := pir;
f2 := pir;
s := f1 [0.5] s := f2;
observe (s = pir)
```

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

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

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

$wlp(P, \mathbf{1}) = 1 - Pr[P \text{ violates an observation}]$. 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: $\dfrac{wp(P, f)}{wlp(P, \mathbf{1})}$

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

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

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

In general:

observe (G) $\equiv$ **while**(~G) **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

```
int x := 1;
while (x = 1) {
    x : = 1
}
```

- Certain divergence
- Conditional termination = 0.

```
int x := 1;
while (x = 1) {
    x : = 1 [0.5] x := 0;
    observe (x = 1)
}
```

- 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 $cwp(P, f) = (wp(P, f), wlp(P, \mathbf{1}))$:

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

Expected rewards in finite Markov chains can be computed in polynomial time.

[1] Either both sides are equal or both sides are undefined.

Overview

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`

```
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:

```
x := 0 [p] x := 1;
y := 0 [p] y := 1;
observe(x =/= y)
```

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

a data-flow analysis yields:

```
x, y := 0, 0;
while(x =/= y) {
  x := 0 [p] x := 1;
  y := 0 [p] y := 1
}
```

Removal of conditioning

Soundness of transformation

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

$$cwp(P, f) \;=\; wp(\widehat{P}, f)$$

A dual program transformation

```
repeat
  a0 := 0 [0.5] a0 := 1;
  a1 := 0 [0.5] a1 := 1;
  a2 := 0 [0.5] a2 := 1;
  i := 4*a0 + 2*a1 + a0 + 1
until (1 <= i <= 6)
```

```
a0 := 0 [0.5] a0 := 1;
a1 := 0 [0.5] a1 := 1;
a2 := 0 [0.5] a2 := 1;
i := 4*a0 + 2*a1 + a0 + 1
observe (1 <= i <= 6)
```

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 $X - Y$

```
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;
      (skip [q] f := 1);
    }
  }
return x;
}
```

Our (semi-automated) analysis yields:

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

Overview

Recursion

Can we also deal with recursion, such as:

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

For instance, with which probability does P terminate?

Recursion

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

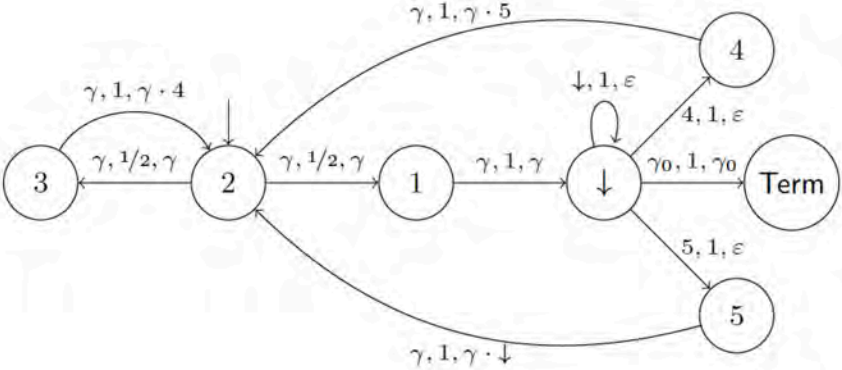
$$\begin{aligned} \texttt{call}_0^D\ P &= \texttt{abort} \\ \texttt{call}_{n+1}^D P &= D(P)[\texttt{call}\ P := \texttt{call}_n^D\ P] \end{aligned}$$

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

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 η equals the expected reward (that depends on f) to reach a terminal state in the pushdown MC of P when starting with η.

Checking expected rewards in finite-control pushdown MDPs is decidable.

Proof rules for recursion

Standard proof rule for recursion:

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

`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 $\ell_0 = \mathbf{0}$ and $u_0 = \mathbf{0}$:

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

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

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

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

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

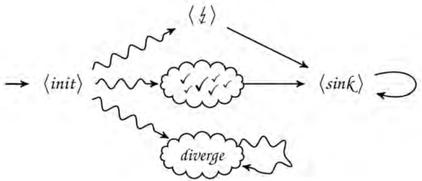
Overview

Non-determinism

[Gordon, Henzinger *et al.* 2014]

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

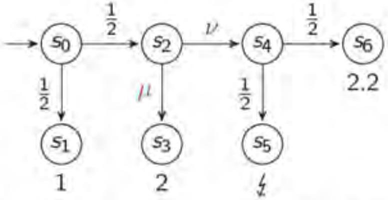
Non-determinism: Operational semantics



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

$$\mathsf{CExpRew}^{\mathfrak{R}}(\Diamond T \mid \neg\Diamond U) \triangleq \inf_{\mathfrak{S} \in Sched^{\mathfrak{R}}} \frac{\mathsf{ExpRew}^{\mathfrak{S}\mathfrak{R}}(\Diamond T \cap \neg\Diamond U)}{\Pr^{\mathfrak{S}\mathfrak{R}}(\neg\Diamond U)}$$

Simple extension. But: conditioning needs policies with memory.



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

Cond. Exp. starting in $s_0 = \frac{1/2 \cdot 1 + 1/4 \cdot 2.2}{3/4} = 1.46$ (taking action ν).

```
x := 1 [1/2] { x := 2 [] { observe(false) [1/2] x := 2.2} }
```

Non-determinism: wp-semantics

Without conditioning:

$$wp(P_1[]P_2, f) \;=\; \min\left(wp(P_1, f), wp(P_2, f)\right)$$

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 $\{P_1\} \Box \{P_2\}$ is an implementation choice. More formally: If it holds that

$$\mathsf{cwp}[\{P_1\} \Box \{P_2\}] = \mathsf{cwp}[P_1]$$

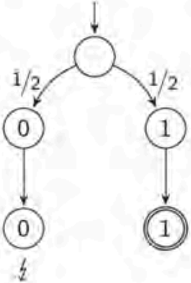
then it should also hold that

$$\mathsf{cwp}[\{\{P_1\} \Box \{P_2\}\} \ [p] \ \{P_3\}] = \mathsf{cwp}[\{P_1\} \ [p] \ \{P_3\}].$$

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.

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

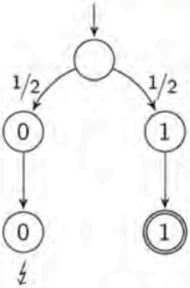


Of course

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

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

$$Q: \quad \{\underbrace{x := 0;\ observe(x = 1)}_{Q_1}\}\ [1/2]\ \{\underbrace{x := 1;\ observe(x = 1)}_{Q_2}\}$$



Of course

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

but we cannot decompose

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

Overview

Termination

[Esparza *et al.* 2012]

"[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

$\Longrightarrow$ almost-sure termination

…… in an expected finite number of steps

$\Longrightarrow$ positive almost-sure termination

…… for all possible program inputs

$\Longrightarrow$ universal [positive] almost-sure termination

Certain termination

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

This program certainly terminates.

Positive almost-sure termination

For p an arbitrary probability:

```
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:

```
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:

```
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 the right after the left program
yields an infinite expected termination time

Three results

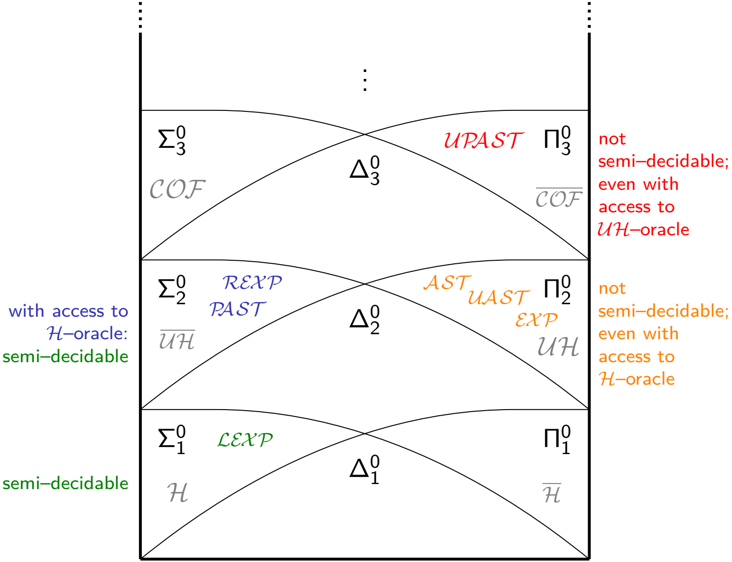
Determining expected outcomes is as hard as almost-sure termination.

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



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 η,
such that P does terminate in an expected finite number of steps on η.

Let's start simple

```
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 $1/2^{\texttt{nrflips}}$.

Reducing an ordinary program to a probabilistic one

Assume an enumeration of all inputs for Q is given

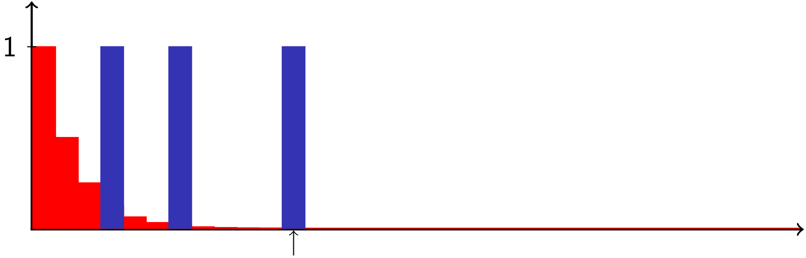
```
bool c := true;
int nrflips := 0;
int i := 0;
while (c) {
    // simulate Q for one (further) step on its i-th input
    if (Q terminates on its ith input) {
        i := i + 1;
        // reset simulation of program Q
        cheer // take 2^nrflips meaningless steps
    } else {
        nrflips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

P looses 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):

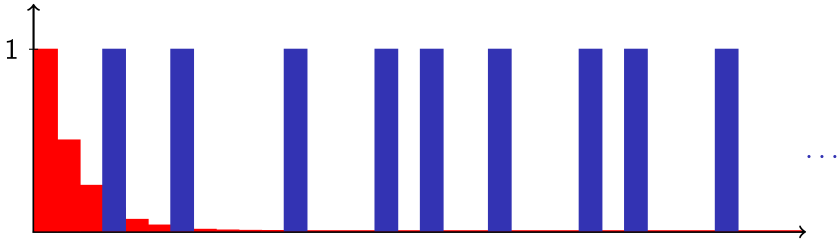


cheering on termination on $(i-1)$-th input

Finite cheering — finite expected runtime!

Q terminates on all inputs

Expected runtime of P (integral over the bars):



Infinite cheering — infinite expected runtime!

Overview

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 $ert() : \mathbb{T} \to \mathbb{T}$ where $\mathbb{T} = \{ t \mid t : \mathcal{S} \to [0, \infty] \}$

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

Expected run-times

Syntax

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

Semantics $ert(P, t)$

- $\mathbf{1} + t$
- $\mathbf{0}$
- $\mathbf{1} + \lambda\sigma. E_{[\![\mu]\!](\sigma)}\left(\lambda v. t[x := v](\sigma)\right)$
- $ert(P_1, ert(P_2, t))$
- $\mathbf{1} + [G] \cdot ert(P_1, t) + [\neg G] \cdot ert(P_2, t)$
- $\max\left(ert(P_1, t), ert(P_2, t)\right)$
- $\mu X. \mathbf{1} + ([G] \cdot ert(P, X) + [\neg G] \cdot t)$

μ is the least fixed point operator wrt. the ordering $\leqslant$ on run-times

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

Coupon collector problem

ON A CLASSICAL PROBLEM OF PROBABILITY THEORY

by

P. ERDŐS and A. RÉNYI

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

```
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

Quantitative loop invariants

Recall that for while-loops we have:

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

To determine this wp, we use an "invariant" I such that $[\neg G] \cdot I \;\leqslant\; f$.

Quantitative loop invariant

Expectation I is a quantitative loop invariant if —by consecution—

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

To guarantee soundness, I has to fulfill either:

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

Then: $\{I\}$ while$(G)\{P\}$ $\{f\}$ is a correct program annotation.

Invariant synthesis for linear programs

inspired by [Colón *et al.* 2002]

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

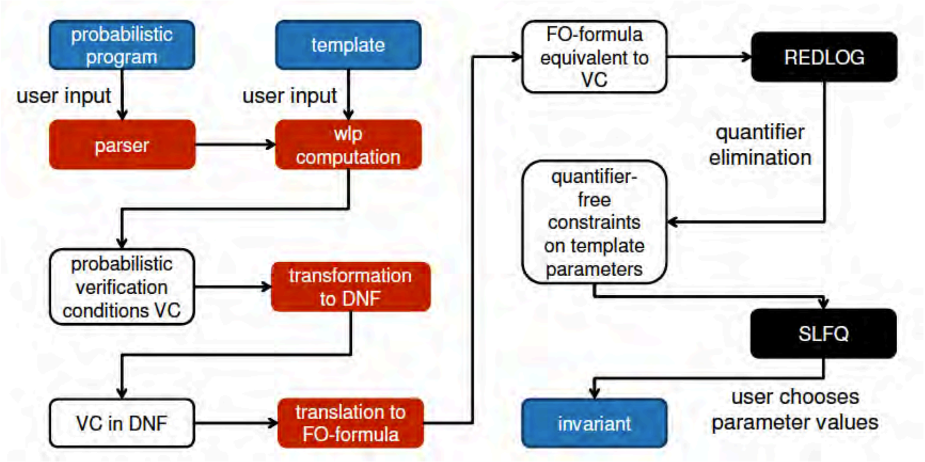
$$[\alpha_1{\cdot}x_1 + \ldots + \alpha_n{\cdot}x_n + \alpha_{n+1} \ll 0] \cdot (\beta_1{\cdot}x_1 + \ldots + \beta_n{\cdot}x_n + \beta_{n+1})$$

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

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



download from `moves.rwth-aachen.de/prinsys`

Program equivalence

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

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

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

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

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

Fin.

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`.
  J. on Performance Evaluation, 2014.
- 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.