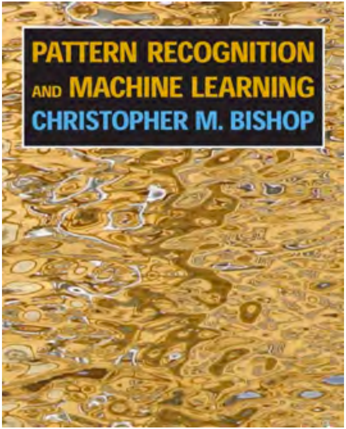
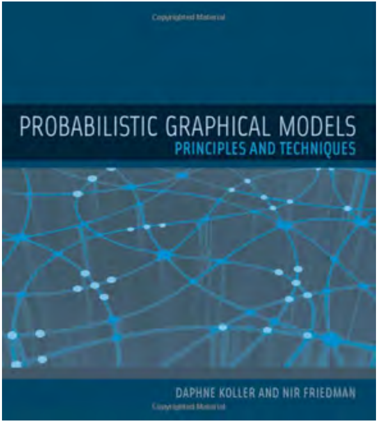
Principles of Probabilistic Programming

Joost-Pieter Katoen

Séminaire de l'IRIF, March 2017

Probabilistic Graphical Models





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

DARPA

DEFENSE ADVANCED RESEARCH PROJECTS AGENCY

Defense Advanced Research Projects Agency › Program Information ›

Probabilistic Programming for Advancing Machine Learning (PPAML)

Probabilistic programs

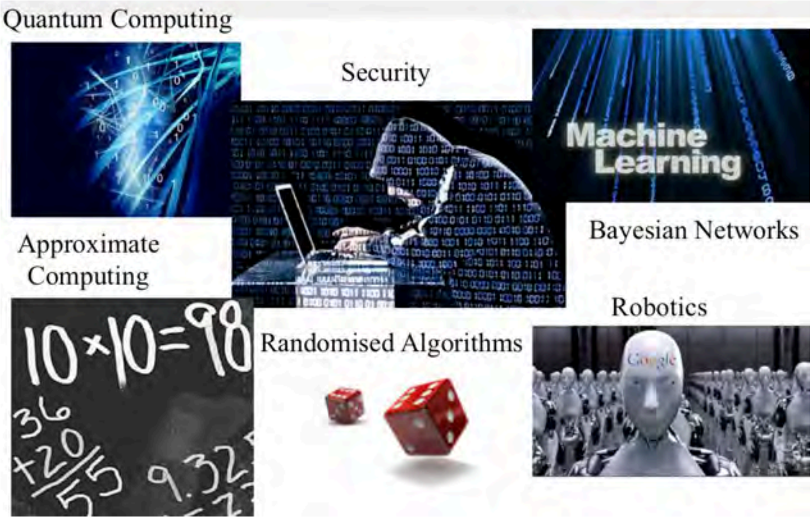
What are probabilistic programs?

Sequential programs with random assignments and conditioning.

[Hicks 2014, The Programming Languages Enthusiast]

"The crux of probabilistic programming is to consider normal-looking programs as if they were probability distributions."

Probabilistic programming applications



Probabilistic programming languages

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

A probabilistic GCL

- `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.

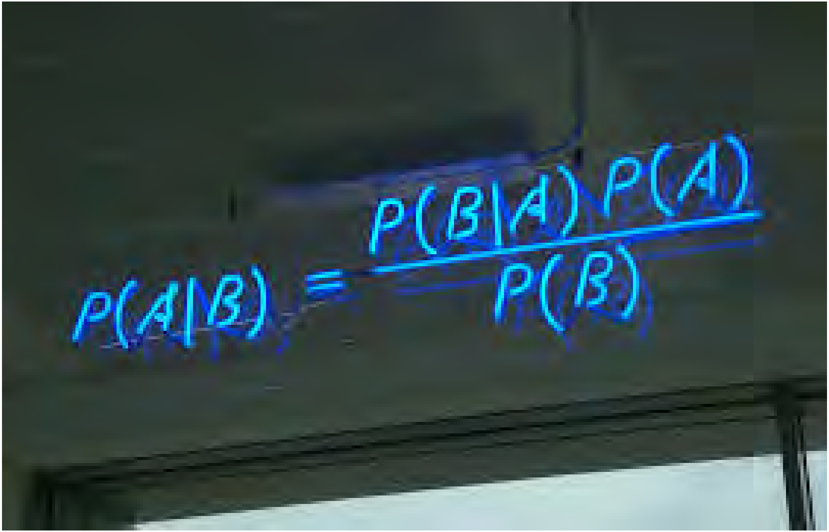
$$Pr[i = N] = (1-p)^{N-1} \cdot p \quad \text{for } N > 0$$

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)
```

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++;
    (c := false [p] c := true)
}
observe (odd(i))
```

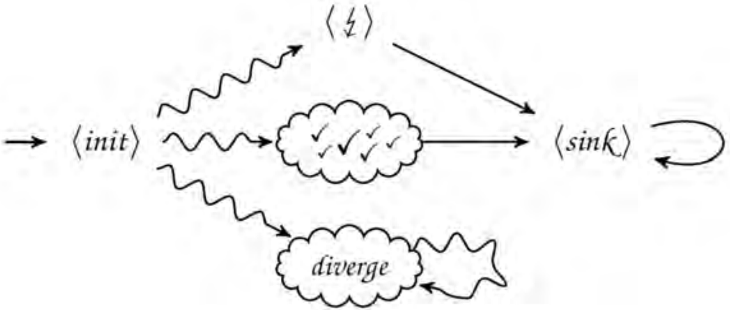
The feasible program runs have a probability $\sum_{N \geq 0} (1-p)^{2N} \cdot p \; = \; \dfrac{1}{2-p}$

This program models the distribution:

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

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

Operational semantics



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

Some inference rules

$$\langle \texttt{skip}, s \rangle \to \langle {\downarrow}, s \rangle \qquad \langle \texttt{abort}, s \rangle \to \langle \texttt{abort}, s \rangle$$

$$\frac{s \models G}{\langle \texttt{observe}(G), s \rangle \to \langle {\downarrow}, s \rangle} \qquad \frac{s \not\models G}{\langle \texttt{observe}(G), s \rangle \to \langle \lightning \rangle}$$

$$\langle {\downarrow}, s \rangle \to \langle sink \rangle \qquad \langle \lightning \rangle \to \langle sink \rangle \qquad \langle sink \rangle \to \langle sink \rangle$$

$$\langle x := E, s \rangle \to \langle {\downarrow}, s[x := s([\![E]\!])] \rangle$$

$$\langle P[\, p\,]\, Q, s \rangle \to \mu \text{ with } \mu(\langle P, s \rangle) = p \text{ and } \mu(\langle Q, s \rangle) = 1{-}p$$

$$\frac{\langle P, s \rangle \to \langle \lightning \rangle}{\langle P; Q, s \rangle \to \langle \lightning \rangle} \qquad \frac{\langle P, s \rangle \to \mu}{\langle P; Q, s \rangle \to \nu} \text{ with } \nu(\langle P'; Q', s' \rangle) = \mu(\langle P', s' \rangle) \text{ where } {\downarrow}; Q = Q$$

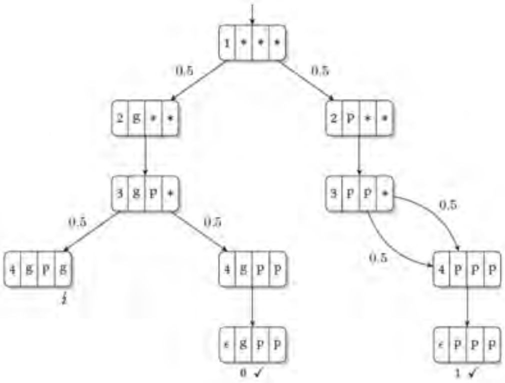
The piranha problem

[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?

The piranha puzzle

```
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

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

Expectations

Weakest pre-expectation [McIver & Morgan 2004]

An expectation[1] maps program states onto non-negative reals. It is the quantitative analogue of a predicate.

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 $e \;\leq\; wp(P, f)$, where $\leq$ is to be interpreted in a point-wise manner.

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

[1] Not to be confused what expectations are in probability theory.

Expectation transformer semantics of `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
- $f[x := E]$
- $[G] \cdot f$
- $wp(P_1, wp(P_2, f))$
- $[G] \cdot wp(P_1, f) + [\neg G] \cdot wp(P_2, f)$
- $p \cdot wp(P_1, f) + (1-p) \cdot wp(P_2, f)$
- $\mu X.\ ([G] \cdot wp(P, X) + [\neg G] \cdot f)$

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

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

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?

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

Let $cwp(P, f) = \dfrac{wp(P, f)}{wlp(P, \mathbf{1})}$. In fact $cwp(P, f) = (wp(P, f), wlp(P, \mathbf{1}))$.

$wlp(P, \mathbf{1}) = 1 - Pr[P$ 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)
}
```

Q: What is the probability that `y = 0` on termination?

We: $\dfrac{wp(P, f)}{wlp(P, \mathbf{1})} = \dfrac{2}{7}$

Microsoft's R2: $\dfrac{wp(P, f)}{wp(P, \mathbf{1})} = \dfrac{2}{3}$

In general:

`observe (G)` $\equiv$ `while(!G) skip`

Warning: This is a silly example. Typically divergence comes from loops.

Leave divergence up to the programmer?

Almost-sure termination is "more undecidable" than ordinary termination!

Observations inside loops

These programs are mostly not distinguished as $wp(P_{left}, \mathbf{1}) = wp(P_{right}, \mathbf{1}) = \mathbf{0}$

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

- Certain divergence
- $(wp(P_{left}, f), wlp(P_{left}, \mathbf{1})) = (\mathbf{0}, \mathbf{1})$
- Conditional wp $= 0$

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

- Divergence with probability zero
- $\big(wp(P_{right}, f), wlp(P_{right}, \mathbf{1})\big) = (\mathbf{0}, \mathbf{0})$
- Conditional wp $=$ undefined

We do distinguish these programs.

Basic properties

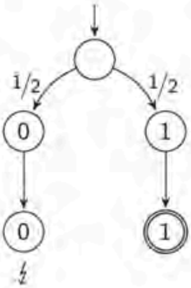
- Monotonicity: $f \leq g$ implies $cwp(P, f) \leq cwp(P, g)$
- Linearity: $cwp(P, \alpha{\cdot}f + \beta{\cdot}g) \;=\; \alpha{\cdot}cwp(P, f) + \beta{\cdot}cwp(P, g)$
- Duality: $cwlp(P, f) \;=\; \mathbf{1} - cwp(P, \mathbf{1}{-}f)$
- Law of excluded miracle: $cwp(P, \mathbf{0}) = \mathbf{0}$

Certified using the Isabelle/HOL theorem prover; see [Hölzl, PPS 2016].

Contextual equivalence?

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

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



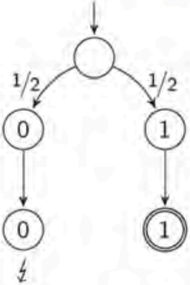
Of course

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

Contextual equivalence?

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

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



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

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

This all motivates the definition: $cwp(P, f) = (wp(P, f), wlp(P, \mathbf{1}))$.

Backward compatibility

McIver's wp-semantics is a conservative extension of Dijkstra's wp-semantics.

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

Wp = 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[2] 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.

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

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
  - ......

Overview

Removal of conditioning

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

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

- Use auxiliary variable `flag` to signal observation violation:

```
flag := true; while(flag)flag := false; modprog
```

- Change `prog` into `modprog` by:
  - `observe(G)` ⇝ `flag := !G && flag`
  - `abort` ⇝ `if(!flag) abort`
  - `while(G) prog` ⇝ `while(G && !flag) prog`

Resulting program

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

In machine learning, this is known as rejection sampling.

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;
  flag := (x = y)
}
```

a simple data-flow analysis yields:

```
repeat {
  x := 0 [p] x := 1;
  y := 0 [p] y := 1
} until(x != y)
```

Removal of conditioning

Correctness of transformation

For program P, transformed program $\widehat{P}$, and 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

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

. . . . . . in an expected infinite number of steps

$\Longrightarrow$ negative almost-sure termination

Certain termination

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

This program certainly terminates.

Positive almost-sure termination

For p 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-- [0.5] x++)
}
```

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--
}
```

Finite termination time

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

Three results

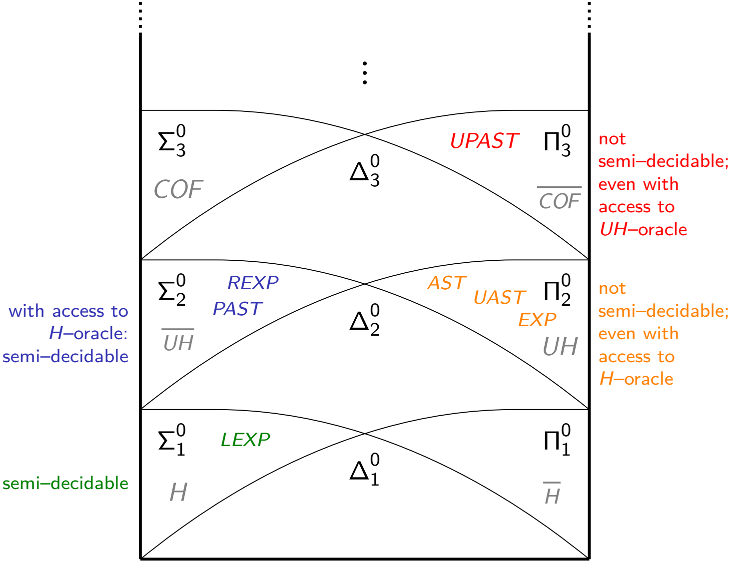
Determining expected outcomes is as hard as almost-sure termination.

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, provide a probabilistic program P (depending on Q) and an input η, such that

P terminates in a finite expected number of steps on η
if and only if
Q does not terminate on some input

Let's start simple

```
bool c := true;
int nrflips := 0;
while (c) {
    nrflips++;
    (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

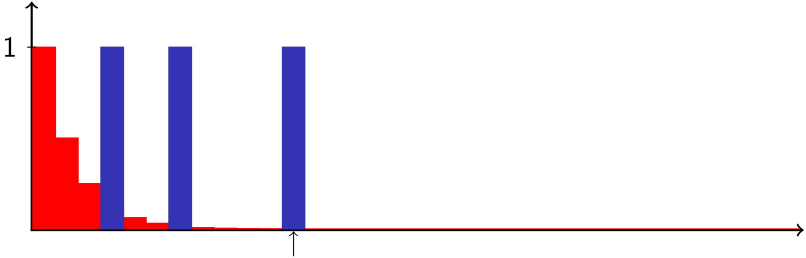
```
bool c := true;
int nrflips := 0;
int i := 0;
while (c) {
    // simulate Q for one (further) step on its i-th input
    if (Q terminates on its i-th input) {
        cheer; // take 2^nrflips effectless steps
        i++;
        // reset simulation of program Q
    }
    nrflips++;
    (c := false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

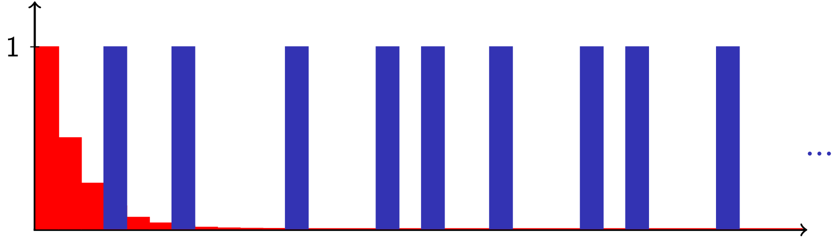


cheering on termination on (i−1)-th input

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. Prove universal positive almost-sure termination $\Rightarrow \Pi^0_3$-complete
2. Reason about the efficiency of randomised algorithms[3]

$ert(P, t)$ bounds P's expected run-time if P's continuation takes t time.

[3] Typically by classical probability theory using martingales and expected values.

A naive, unsound approach

Equip the program with a counter `rc` and use standard wp-reasoning.

```
rc : = 0;
c := false [0.5] c := true; rc++;
for (i := 1; i < 2k-1; i++) // 2k-1 useless steps
       { skip; rc++ }
while (c) { skip; rc++ }
```

The expected value of `rc` is k, but the expected run-time is ∞

Expected run-times

Syntax	Semantics $ert(P, t)$
`skip`	$\mathbf{1} + t$
`abort`	∞
`x := mu`	$\mathbf{1} + \lambda\sigma.E_{[\![\,\mu\,]\!](\sigma)}\,(\lambda v.t[x := v](\sigma))$
`observe (G)`	$[G] \cdot (\mathbf{1} + t)$
`P1 ; P2`	$ert(P_1, ert(P_2, t))$
`if (G)P1 else P2`	$\mathbf{1} + [G] \cdot ert(P_1, t) + [\neg G] \cdot ert(P_2, t)$
`while(G)P`	$\mu X.\, \mathbf{1} + ([G] \cdot ert(P, X) + [\neg G] \cdot t)$

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

and a set of proof rules [4] to get two-sided bounds on run-times of loops

[4] Certified using the Isabelle/HOL theorem prover; see [Hölzl, ITP 2016].

Proof rules for loops

Let n be a natural and let $\texttt{while}(G)\,P$ be our loop.

Run-time transformer I_n is a lower ω-invariant w.r.t. t iff

$$I_0 \preceq F_t(\mathbf{0}) \quad \text{and} \quad I_{n+1} \preceq F_t(I_n) \text{ for all } n$$

where $F_t(X) = \mu X.\, \mathbf{1} + ([G] \cdot ert(P, X) + [\neg G] \cdot t)$.

In a similar way, upper ω-invariants w.r.t. t are defined.

If I_n is a lower ω-invariant w.r.t. t and $\lim_{n\to\infty} I_n$ exists, then:

$$\lim_{n\to\infty} I_n \preceq ert(\texttt{while}(G)\,P, t)$$

Upper ω-invariants provide an upper bound on the loop's run time.

Completeness: such lower- and upper ω-invariants always exist.

Invariant synthesis

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

Template for a lower ω-invariant:

$$I_n \;=\; \mathbf{1} \;+\; \underbrace{[c \neq 1] \cdot (\mathbf{1} + [x > 0]{\cdot}2x)}_{\text{on termination}} \;+\; \underbrace{[c = 1] \cdot (a_n + b_n \cdot [x > 0]{\cdot}2x)}_{\text{on iteration}}$$

The constraints on being a lower ω-invariant yield:

$$a_0 \;\leq\; 2 \quad \text{and} \quad a_{n+1} \;\leq\; 7/2 + 1/2{\cdot}a_n \quad \text{and} \quad b_0 \;\leq\; 0 \quad \text{and} \quad b_{n+1} \;\leq\; 1 + b_n$$

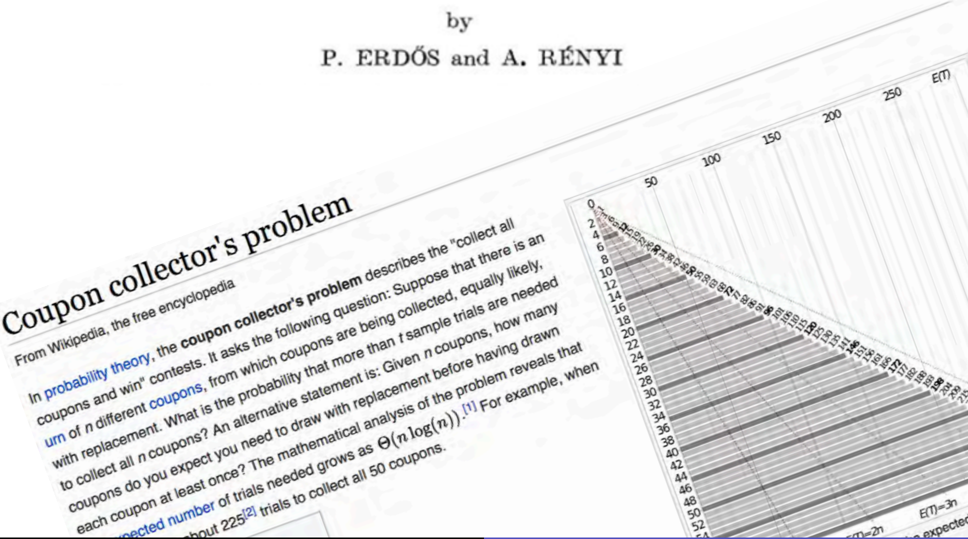
This admits the solution $a_n = 7 - 5/2^n$ and $b_n = n$.

Coupon collector's problem

ON A CLASSICAL PROBLEM OF PROBABILITY THEORY

by

P. ERDŐS and A. RÉNYI



Coupon collector's 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?

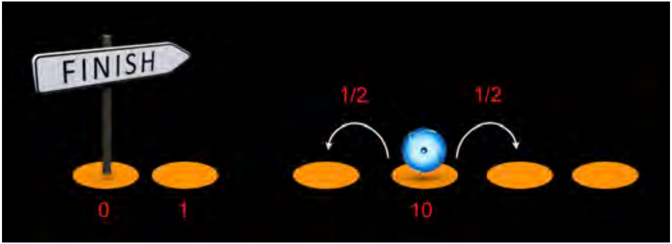
Coupon collector's problem

```
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
}
```

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. Machine checkable.

Random walk



Using our ert-calculus one can prove that its expected run-time is ∞.
By systematic formal verification à la Floyd-Hoare. Machine checkable.

Overview

Recursion

Q: What is the probability that recursive program 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:

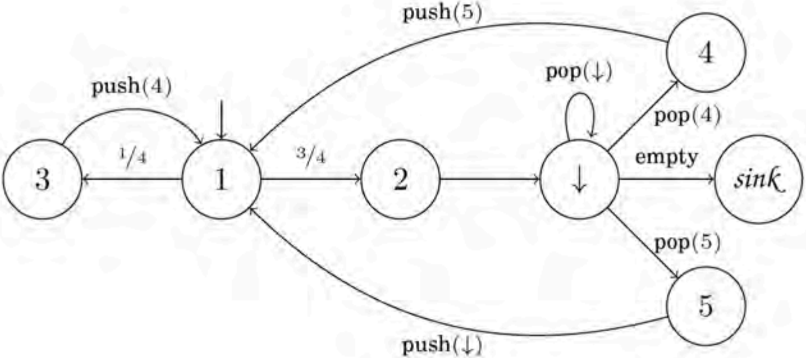
$$\begin{aligned}\mathtt{call}_0^D\ P &= \mathtt{abort}\\ \mathtt{call}_{n+1}^D P &= D(P)[\mathtt{call}\ P := \mathtt{call}_n^D\ P]\end{aligned}$$

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

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

This corresponds to the fixed point of a (higher order) environment transformer

Pushdown Markov chains



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

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 MCs is decidable.[5]

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

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

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

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

$\triangle$

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 $X - Y$

```
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;
}
```

```
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:

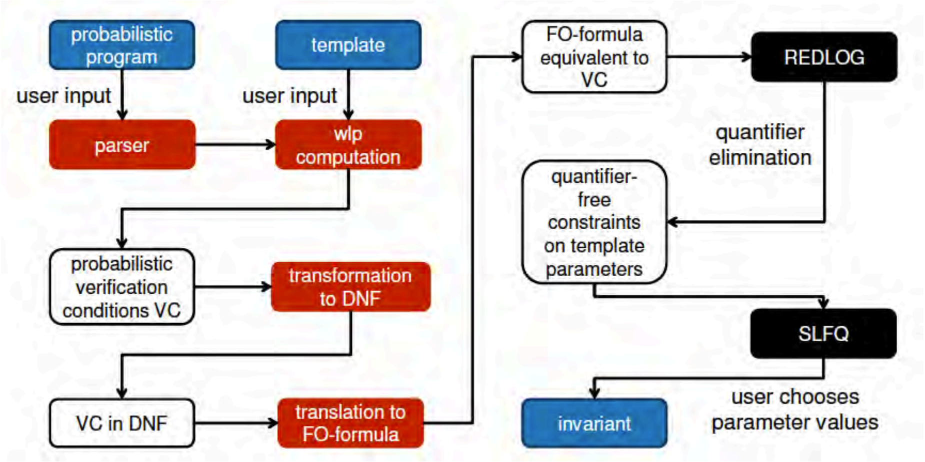
$$[\alpha_1 \cdot x_1 + \ldots + \alpha_n \cdot x_n + \alpha_{n+1} \ll 0] \cdot (\beta_1 \cdot x_1 + \ldots + \beta_n \cdot x_n + \beta_{n+1})$$

   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 will find all parameter solutions that make the template 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++ [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 $\alpha_{22} = -\frac{1}{1-q}$.

Epilogue

Take-home message

- Connecting wp and operational semantics
- Semantic intricacies of conditioning
- Almost-sure termination is harder than termination
- Expected run-time analysis

Extensions

- Non-determinism
- Mixed-sign random variables
- Link to Bayesian networks
- Invariant synthesis

Fin.

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. Gretz *et al.*.
  Conditioning in probabilistic programming. MFPS 2015.
- 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 [6]*.* ESOP 2016.
- F. Olmedo, B. Kaminski, JPK, C. Matheja.
  Reasoning about recursive probabilistic programs. LICS 2016.

[6] Recipient EATCS best paper award of ETAPS 2016.