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”

aMIT/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.

[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

Languages:
Probabilistic C
ProbLog
Church
webPPL
Figaro
PyMC
Tabular
R2

probabilistic-programming.org

A. Pfeffer

N. Goodman

Microsoft Research
Roadmap of this talk

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

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

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

empty statement
abortion
assignment
conditioning
sequential composition
choice
probabilistic choice
iteration
Let’s start simple

\[
\begin{align*}
    x &:= 0 \ [0.5] \ x := 1; \\
    y &:= -1 \ [0.5] \ y := 0 \\
\end{align*}
\]

This program admits four runs and yields the outcome:

\[
\begin{align*}
    Pr[x=0, y=0] &= Pr[x=0, y=-1] = Pr[x=1, y=0] = Pr[x=1, y=-1] = \frac{1}{4}
\end{align*}
\]
A loopy program

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

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

The loopy program models a geometric distribution with parameter $p$.

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

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

\[
x := 0 [0.5] \quad x := 1;
\]
\[
y := -1 [0.5] \quad y := 0;
\]
\[
\text{observe} \ (x+y = 0)
\]

This program blocks two runs as they violate \(x+y = 0\). Outcome:

\[
Pr[x=0, y=0] = Pr[x=1, y=-1] = \frac{1}{2}
\]

Observations thus normalize the probability of the “feasible” program runs.
A loopy program

For \( p \) an arbitrary probability:

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

The feasible program runs have a probability

\[
\sum_{N \geq 0} (1-p)^{2N} \cdot p = \frac{1}{2-p}
\]

This program models the distribution:

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

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

This can be defined using Plotkin’s SOS-style semantics
Some inference rules

\[ \langle \text{skip}, s \rangle \rightarrow \langle \downarrow, s \rangle \quad \langle \text{abort}, s \rangle \rightarrow \langle \text{abort}, s \rangle \]

\[
\begin{align*}
\frac{s \models G}{\langle \text{observe}(G), s \rangle \rightarrow \langle \downarrow, s \rangle} & \quad \frac{s \not\models G}{\langle \text{observe}(G), s \rangle \rightarrow \langle \top \rangle} \\
\langle \downarrow, s \rangle \rightarrow \langle \text{sink} \rangle & \quad \langle \top \rangle \rightarrow \langle \text{sink} \rangle & \quad \langle \text{sink} \rangle \rightarrow \langle \text{sink} \rangle \\
\langle x := E, s \rangle & \rightarrow \langle \downarrow, s[x := s[\llbracket E \rrbracket]] \rangle \\
\end{align*}
\]

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

\[
\frac{\langle P, s \rangle \rightarrow \langle \top \rangle}{\langle P; Q, s \rangle \rightarrow \langle \top \rangle} \quad \frac{\langle P, s \rangle \rightarrow \mu}{\langle P; Q, s \rangle \rightarrow \nu} \text{ with } \nu(\langle P'; Q', s' \rangle) = \mu(\langle P', s' \rangle) \text{ where } \downarrow; Q = Q
\]
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?
The piranha puzzle

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

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

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

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

\(^1\)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)`

\( \mu \) is the least fixed point operator wrt. the ordering \( \leq \) on expectations.

wlp-semantics differs from wp-semantics only for `while` and `abort`. 
\[
\begin{align*}
    x & := 0 \ [1/2] \ x := 1; \ // \ \text{command } c1 \\
y & := 0 \ [1/3] \ y := 1; \ // \ \text{command } c2
\end{align*}
\]

\[
wp(c_1; c_2, [x = y])
= wp(c_1, wp(c_2, [x = y]))
= wp(c_1, 1/3 \cdot wp(y := 0, [x = y]) + 2/3 \cdot wp(y := 1, [x = y]))
= wp(c_1, 1/3 \cdot [x = 0] + 2/3 \cdot [x = 1])
= 1/2 \cdot wp(x := 0, 1/3 \cdot [x = 0] + 2/3 \cdot [x = 1]) + 1/2 \cdot wp(x := 1, 1/3 \cdot [x = 0] + 2/3 \cdot [x = 1])
= 1/2 \cdot (1/3 \cdot [0 = 0] + 2/3 \cdot [0 = 1]) + 1/2 \cdot (1/3 \cdot [1 = 0] + 2/3 \cdot [1 = 1])
= 1/2 \cdot (1/3 \cdot 1 + 2/3 \cdot 0) + 1/2 \cdot (1/3 \cdot 0 + 2/3 \cdot 1)
= 1/2 \cdot (1/3 + 2/3)
= 1/2
\]
The piranha program – a wp perspective

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

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

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

Let \( cwP(P, f) = \frac{wp(P, f)}{wlP(P, 1)} \). In fact \( cwP(P, f) = (wp(P, f), wlP(P, 1)) \).

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

\[
\text{abort} \ [0.5] \{
\begin{array}{ll}
x := 0 \ [0.5] & x := 1; \\
y := 0 \ [0.5] & y := 1;
\end{array}
\text{observe} \ (x = 0 \mid\mid y = 0)
\}
\]

Q: What is the probability that \( y = 0 \) on termination?

\[
\begin{align*}
\text{We:} & \quad \frac{wp(P, f)}{wlP(P, 1)} = \frac{2}{7} \\
\text{Microsoft's R2:} & \quad \frac{wp(P, f)}{wp(P, 1)} = \frac{2}{3} \\
\text{In general:} & \quad \text{observe } (G) \equiv \text{while}(!G) \text{ skip}
\end{align*}
\]

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 \( wp(P_{left}, 1) = wp(P_{right}, 1) = 0 \)

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

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

We do distinguish these programs.
Basic properties

- **Monotonicity**: \( f \leq g \) implies \( \text{cwp}(P, f) \leq \text{cwp}(P, g) \)

- **Linearity**: \( \text{cwp}(P, \alpha \cdot f + \beta \cdot g) = \alpha \cdot \text{cwp}(P, f) + \beta \cdot \text{cwp}(P, g) \)

- **Duality**: \( \text{cwlp}(P, f) = 1 - \text{cwp}(P, 1-f) \)

- **Law of excluded miracle**: \( \text{cwp}(P, 0) = 0 \)

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

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

Of course

\[ \frac{wp(P, [x = 1])}{wlP(P, 1)} = \frac{wp(Q, [x = 1])}{wlP(Q, 1)} = \frac{1}{2} \frac{1}{2} = 1 \]
Contextual equivalence?

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

The ratio of $wp(P, f)$ over $wlp(P, 1)$ for input $\eta$ equals\(^2\) the conditional expected reward to reach a successful terminal state in $P$’s MC when starting with $\eta$.

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

1. Introduction
2. Two flavours of semantics
3. Program transformations
4. Different flavours of termination
5. Run-time analysis
6. Recursion
7. Synthesizing loop invariants
8. Epilogue
Removal of conditioning

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

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

- Use auxiliary variable $flag$ to signal observation violation:
  
  \[
  \text{flag := true; while}(\text{flag}) \text{flag := false}; \text{modprog}
  \]

- Change prog into modprog by:
  
  - \(\text{observe}(G) \implies \text{flag := !G && flag}\)
  - \(\text{abort} \implies \text{if}(!\text{flag}) \text{abort}\)
  - \(\text{while}(G) \text{ prog} \implies \text{while}(G \&\& !\text{flag}) \text{ prog}\)
Resulting program

\[
s_{x_1}, \ldots, s_{x_n} := x_1, \ldots, x_n; \text{ flag := true;}
\]
\[
\text{while (flag)} \{ \\
    \text{flag := false;}
    x_1, \ldots, x_n := s_{x_1}, \ldots, s_{x_n};
    \text{modprog}
\}
\]

In machine learning, this is known as rejection sampling.
Removal of conditioning

the transformation in action:

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

a simple data-flow analysis yields:

\[
\begin{align*}
\text{repeat } & \{ \\
\text{\quad x := 0 [p] x := 1; } \\
\text{\quad y := 0 [p] y := 1 } \\
\text{\} until(x \neq y) } \\
\end{align*}
\]
Removal of conditioning

Correctness of transformation

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

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

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

\[ \text{a0 := 0 [0.5] a0 := 1;} \]
\[ \text{a1 := 0 [0.5] a1 := 1;} \]
\[ \text{a2 := 0 [0.5] a2 := 1;} \]
\[ \text{i := 4*a0 + 2*a1 + a0 + 1} \]
\[ \text{observe (1 \leq i \leq 6)} \]

Loop-by-observe replacement if there is “no data flow” between loop iterations
Overview

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

“[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.”

[Esparza et al. 2012]
Nuances of termination

...... certain termination

...... termination with probability one
        \implies \text{almost-sure termination}

...... in an expected finite number of steps
        \implies \text{positive almost-sure termination}

...... in an expected infinite number of steps
        \implies \text{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:

```c
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:

```java
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:

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

Finite expected termination time

```c
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 \textit{as hard as} almost-sure termination.

Almost-sure termination is \textquoteleft more undecidable\textquoteright{} than ordinary termination.

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

- **Σ₀¹**: semi-decidable with access to $H$–oracle.
- **Π₀¹**: not semi-decidable; even with access to $UH$–oracle.
- **Δ₀¹**: not semi-decidable; even with access to $H$–oracle.

**Notes**:
- **COF**: with access to $H$–oracle: semi-decidable.
- **UPAST**: not semi-decidable; even with access to $UH$–oracle.

**Abbreviations**:
- **REXP**: REEXP
- **PAST**: PAST
- **AST**: AST
- **UAST**: UAST
- **EXP**: EXP
- **H**: H
- **UH**: UH
- **COF**: COF
- **PAST**: PAST
- **AST**: AST
- **UAST**: UAST
- **EXP**: EXP
- **H**: H
- **UH**: UH
- **COF**: COF
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 $\eta$, such that

$P$ terminates in a finite expected number of steps on $\eta$

if and only if

$Q$ does not terminate on some input
Let’s start simple

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

The expected runtime (integral over the bars):

\[
\text{Expected runtime (integral over the bars)}:
\]

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) {
        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):

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
4. Different flavours of termination
5. Run-time analysis
6. Recursion
7. Synthesizing loop invariants
8. Epilogue
Expected run-times

Aim

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

1. Prove universal positive almost-sure termination $\Rightarrow \Pi_3^0$-complete
2. Reason about the efficiency of randomised algorithms

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

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 $\infty$
**Expected run-times**

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Semantics $ert(P, t)$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$1 + t$</td>
</tr>
<tr>
<td>skip</td>
<td>$\infty$</td>
</tr>
<tr>
<td>abort</td>
<td>$1 + \lambda \sigma. E_{\mu}(\sigma) (\lambda v. t<a href="%5Csigma">x := v</a>)$</td>
</tr>
<tr>
<td>$x := \text{mu}$</td>
<td>$[G] \cdot (1 + t)$</td>
</tr>
<tr>
<td>observe (G)</td>
<td>$ert(P_1, ert(P_2, t))$</td>
</tr>
<tr>
<td>P1 ; P2</td>
<td>$1 + [G] \cdot ert(P_1, t) + [\neg G] \cdot ert(P_2, t)$</td>
</tr>
<tr>
<td>if (G)P1 else P2</td>
<td>$\mu X. 1 + ([G] \cdot ert(P, X) + [\neg G] \cdot t)$</td>
</tr>
<tr>
<td>while(G)P</td>
<td></td>
</tr>
</tbody>
</table>

$\mu$ 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 $\text{while}(G) P$ be our loop.

Run-time transformer $I_n$ is a **lower $\omega$-invariant** w.r.t. $t$ iff

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

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

In a similar way, **upper $\omega$-invariants** w.r.t. $t$ are defined.

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

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

**Upper $\omega$-invariants** provide an upper bound on the loop’s run time.

**Completeness**: such lower- and upper $\omega$-invariants always exist.
Invariant synthesis

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

Template for a lower $\omega$-invariant:

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

The constraints on being a lower $\omega$-invariant yield:

\[
a_0 \leq 2 \quad \text{and} \quad a_{n+1} \leq \frac{7}{2} + \frac{1}{2} \cdot a_n \quad \text{and} \quad b_0 \leq 0 \quad \text{and} \quad b_{n+1} \leq 1 + b_n
\]

This admits the solution $a_n = 7 - \frac{5}{2^n}$ and $b_n = n$. 
Coupon collector’s problem

From Wikipedia, the free encyclopedia

In probability theory, the **coupon collector's problem** describes the “collect all coupons and win” contests. It asks the following question: Suppose that there is an urn of \( n \) different coupons, from which coupons are being collected, equally likely, with replacement. What is the probability that more than \( t \) sample trials are needed to collect all \( n \) coupons? An alternative statement is: Given \( n \) coupons, how many coupons do you expect you need to draw with replacement before having drawn each coupon at least once? The mathematical analysis of the problem reveals that the expected number of trials needed grows as \( \Theta(n \log(n)) \). \[^1\] For example, when about 225\[^2\] trials to collect all 50 coupons.

\[^2\] For n = 50, the expected number of trials is approximately 225.
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

```plaintext
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 $\infty$. By systematic formal verification à la Floyd-Hoare. Machine checkable.
Overview

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

\[
\begin{align*}
\text{call}_0^D P &= \text{abort} \\
\text{call}_{n+1}^D P &= D(P)[\text{call } P := \text{call}_n^D P]
\end{align*}
\]

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

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

\[
\{\text{skip}^1\} \left[ \frac{1}{2} \right]^2 \{\text{call } P^3; \text{ call } P^4; \text{ call } P^5 \}\]

Wₚ = expected rewards in pushdown MCs

For recursive program \( P \) and post-expectation \( f \):

\[ wp(P, f) \]

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

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:

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

\(\text{call } P \) 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 \( \ell_0 = 0 \) and \( u_0 = 0 \):

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

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

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

terminates with probability \[ \frac{\sqrt{5} - 1}{2} = \frac{1}{\phi} = \phi \]

Or: apply to reason about Sherwood variants of binary search, quick sort etc.
\[
\begin{align*}
wp[\text{call } P](1) & \leq \varphi \quad \dashv \quad wp[\mathcal{D}(P_{rec3})](1) \leq \varphi \\
wp[\mathcal{D}(P_{rec3})](1) &= \{\text{def. of wp}\} \\
&= \frac{1}{2} \cdot \wp[\text{skip}](1) + \frac{1}{2} \cdot \wp[\text{call } P_{rec3} ; \text{ call } P_{rec3} ; \text{ call } P_{rec3}](1) \\
&= \{\text{def. of wp}\} \\
&= \frac{1}{2} + \frac{1}{2} \cdot \wp[\text{call } P_{rec3} ; \text{ call } P_{rec3}](wp[\text{call } P_{rec3}](1)) \\
&= \{\text{assumption, monot. of wp}\} \\
&= \frac{1}{2} + \frac{1}{2} \varphi \cdot \wp[\text{call } P_{rec3}](wp[\text{call } P_{rec3}](1)) \\
&= \{\text{assumption, monot. of wp}\} \\
&= \frac{1}{2} + \frac{1}{2} \varphi \cdot \wp[\text{call } P_{rec3}](\varphi) \\
&= \{\text{scalab. of wp}\} \\
&= \frac{1}{2} + \frac{1}{2} \varphi^2 \cdot \wp[\text{call } P_{rec3}](1) \\
&= \{\text{assumption, monot. of wp}\} \\
&= \frac{1}{2} + \frac{1}{2} \varphi^3 \\
&= \{\text{algebra}\} \\
\varphi \\
\end{align*}
\]
Overview

1 Introduction
2 Two flavours of semantics
3 Program transformations
4 Different flavours of termination
5 Run-time analysis
6 Recursion
7 **Synthesizing loop invariants**
8 Epilogue
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 (!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

```c
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;
}
```

```c
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:

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

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., 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**

- **Parser** takes user input of a probabilistic program.
- **WLP Computation** transforms the program into a template.
- **Transformation to DNF** converts the template into Disjunctive Normal Form (DNF).
- **FO-formula Equivalent to VC** generates a first-order formula equivalent to the verification condition (VC).
- **Redlog** performs quantifier elimination on the FO-formula.
- **SLFQ** further processes the quantifier-free constraints on template parameters.
- **User Chooses Parameter Values** to synthesize the invariant.

Download from [moves.rwth-aachen.de/prinsys](http://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:

\[
\alpha_{11} = \frac{p}{1-p}, \quad \alpha_{12} = -\frac{q}{1-q}, \quad \alpha_{21} = \alpha_{11} \quad \text{and} \quad \alpha_{22} = -\frac{1}{1-q}.
\]
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
Further reading

- JPK, A. McIver, L. Meinicke, and C. Morgan. 

- F. Gretz, JPK, and A. McIver. 

- 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.

- F. Olmedo, B. Kaminski, JPK, C. Matheja.
  *Reasoning about recursive probabilistic programs.* LICS 2016.

---

6 Recipient EATCS best paper award of ETAPS 2016.