Weakest Precondition Reasoning for Expected Run–Times of Probabilistic Programs

Benjamin Kaminski    Joost-Pieter Katoen
Christoph Matheja    Federico Olmedo

25th European Symposium on Programming
19th edition of the European Joint Conferences on Theory & Practice of Software
April 4, 2016, Eindhoven, Netherlands
Probabilistic Programs

Motivation

Introduce randomization into computation. Significant speed-up in solving difficult problems at cost of tolerating incorrect results with low probability.

Solution to problems where deterministic techniques fail: E.g. symmetry breaking in Dining Philosophers, Leader Election, Ethernet's randomized exponential backoff.

Randomization of some sort occurs almost in any technique related used in cryptography and security.

Model probability distributions in machine learning.

Kaminski, Katoen, Matheja, Olmedo

Weakest Precondition Reasoning for Expected Run–Times

4.4.2016
Probabilistic Programs

- Introduce randomization into computation
Probabilistic Programs

- Introduce randomization into computation
- Significant speed–up in solving difficult problems at cost of tolerating incorrect results with low probability
Probabilistic Programs

- Introduce *randomization* into computation
- Significant *speed–up in solving difficult problems* at cost of tolerating incorrect results with low probability
- Solution to problems *where deterministic techniques fail*:
  
  E.g. *symmetry breaking* in Dining Philosophers, Leader Election, Ethernet’s randomized exponential backoff
Probabilistic Programs

- Introduce randomization into computation
- Significant speed–up in solving difficult problems at cost of tolerating incorrect results with low probability
- Solution to problems where deterministic techniques fail:
  - E.g. symmetry breaking in Dining Philosophers, Leader Election, Ethernet’s randomized exponential backoff
- Randomization of some sort occurs almost in any technique related used in cryptography and security
Probabilistic Programs

- Introduce randomization into computation
- Significant speed–up in solving difficult problems at cost of tolerating incorrect results with low probability
- Solution to problems where deterministic techniques fail:
  - E.g. symmetry breaking in Dining Philosophers, Leader Election, Ethernet’s randomized exponential backoff
- Randomization of some sort occurs almost in any technique related used in cryptography and security
- Model probability distributions in machine learning
Syntax of Probabilistic Programs

\[
C \quad \rightarrow \quad \text{skip} \quad | \quad x := E \quad | \quad C; \ C \quad | \quad \{C\} \Box \{C\} \\
| \quad \text{if} (\xi) \{C\} \text{ else } \{C\} \quad | \quad \text{while} (\xi) \{C\}
\]
## Syntax of Probabilistic Programs

\[
C \quad \rightarrow \quad \text{skip} \quad | \quad x := E \quad | \quad C; C \quad | \quad \{C\} \square \{C\} \\
| \quad \text{if } (\xi) \{C\} \text{ else } \{C\} \quad | \quad \text{while } (\xi) \{C\}
\]
### Syntax of Probabilistic Programs

\[
C \quad \rightarrow \quad \text{skip} \quad | \quad x := E \quad | \quad C; C \quad | \quad \{C\} \square \{C\} \\
| \quad \text{if } (\xi) \{C\} \text{ else } \{C\} \quad | \quad \text{while } (\xi) \{C\}
\]
### Syntax of Probabilistic Programs

\[
C \quad \rightarrow \quad \text{skip} \mid x := E \mid C; C \mid \{C\} \, \square \{C\} \\
\quad \mid \text{if } (\xi) \{C\} \text{ else } \{C\} \mid \text{while } (\xi) \{C\}
\]
Syntax of Probabilistic Programs

\[ C \quad \rightarrow \quad \text{skip} \mid x := E \mid C; C \mid \{C\} \Box \{C\} \]
\[ \mid \text{if } (\xi) \{C\} \text{ else } \{C\} \mid \text{while } (\xi) \{C\} \]
Syntax of Probabilistic Programs

\[
C \rightarrow \text{skip} \mid x := E \mid C; C \mid \{C\} \square \{C\}
\]
\[
\mid \text{if} (\xi) \{C\} \text{ else } \{C\} \mid \text{while} (\xi) \{C\}
\]
Syntax of Probabilistic Programs

\[ C \quad \rightarrow \quad \text{skip} \quad | \quad x := E \quad | \quad C; \ C \quad | \quad \{ C \} \ □ \ \{ C \} \quad | \quad \text{if} (\xi) \{ C \} \ \text{else} \{ C \} \quad | \quad \text{while} (\xi) \{ C \} \]
Syntax of Probabilistic Programs

\[
C \quad \rightarrow \quad \text{skip} \mid x := E \mid C; C \mid \{C\} \diamond \{C\} \\
\quad \mid \text{if}(\xi)\{C\} \text{ else } \{C\} \mid \text{while}(\xi)\{C\}
\]

What is probabilistic about that language?
Motivation

Probabilistic Programs

Syntax of Probabilistic Programs

\[
C \quad \rightarrow \quad \text{skip} \quad | \quad x := E \quad | \quad C; \ C \quad | \quad \{C\} \ \square \ \{C\} \\
\quad | \quad \text{if} (\xi) \ \{C\} \ \text{else} \ \{C\} \quad | \quad \text{while} (\xi) \ \{C\}
\]

What is probabilistic about that language?

Probabilistic guards \( \xi : \Sigma \rightarrow \mathcal{D}({\text{true, false}}) \):
Syntax of Probabilistic Programs

\[
C \quad \rightarrow \quad \text{skip} \mid x := E \mid C; C \mid \{C\} \sqcap \{C\} \mid \text{if} (\xi) \{C\} \text{ else } \{C\} \mid \text{while} (\xi) \{C\}
\]

What is probabilistic about that language?

**Probabilistic guards** $\xi : \Sigma \rightarrow \mathcal{D}(\{\text{true, false}\})$:

- $\llbracket \xi : \text{true} \rrbracket(\sigma) = 1 - \llbracket \xi : \text{false} \rrbracket(\sigma)$ is the probability of $\xi$ evaluating to true
Motivation

Probabilistic Programs

Syntax of Probabilistic Programs

\[ C \rightarrow \text{skip} \mid x := E \mid C; C \mid \{C\} \square \{C\} \]
\[ \mid \text{if } (\xi) \{C\} \text{ else } \{C\} \mid \text{while } (\xi) \{C\} \]

What is probabilistic about that language?

**Probabilistic guards** \( \xi : \Sigma \rightarrow D(\{\text{true, false}\}) : \)

- \( \llbracket \xi : \text{true} \rrbracket (\sigma) = 1 - \llbracket \xi : \text{false} \rrbracket (\sigma) \) is the probability of \( \xi \) evaluating to true
- E.g. \( \frac{2}{3} \langle \text{true} \rangle + \frac{1}{3} \langle \text{false} \rangle \)
Syntax of Probabilistic Programs

\[
C \quad \rightarrow \quad \text{skip} \quad | \quad x := E \quad | \quad C; \ C \quad | \quad \{C\} \square \{C\} \\
\quad | \quad \text{if } (\xi) \{C\} \quad \text{else } \{C\} \quad | \quad \text{while } (\xi) \{C\}
\]

What is probabilistic about that language?

Probabilistic guards \(\xi: \Sigma \rightarrow \mathcal{D}(\{\text{true, false}\})\):

- \([\xi: \text{true}] (\sigma) = 1 - [\xi: \text{false}] (\sigma)\) is the probability of \(\xi\) evaluating to true

- E.g. \(\frac{2}{3} \langle \text{true} \rangle + \frac{1}{3} \langle \text{false} \rangle, \quad \frac{1}{2} \langle x > y \rangle + \frac{1}{2} \langle x \geq y \rangle\)
Probabilistic Programs

What does a probabilistic program $C$ do?
Probabilistic Programs

What does a probabilistic program $C$ do?

- Run program $C$ on initial state $\sigma$
Probabilistic Programs

What does a probabilistic program $C$ do?

- Run program $C$ on initial state $\sigma$
- Obtain final set of distributions $\mu$ over terminal states
Probabilistic Programs

What does a probabilistic program $C$ do?

- Run program $C$ on initial state $\sigma$
- Obtain final set of (sub–)distributions $\mu$ over terminal states
Probabilistic Programs

What does a probabilistic program $C$ do?

- Run program $C$ on initial state $\sigma$
- Obtain final set of (sub–)distributions $\mu$ over terminal states

What is the run–time of $C$ on input $\sigma$?
Probabilistic Programs

What does a probabilistic program $C$ do?

- Run program $C$ on initial state $\sigma$
- Obtain final set of (sub–)distributions $\mu$ over terminal states

What is the run–time of $C$ on input $\sigma$?

- Behavior of $C$ not entirely determined by $\sigma$
Probabilistic Programs

What does a probabilistic program $C$ do?

- Run program $C$ on initial state $\sigma$
- Obtain final set of (sub–)distributions $\mu$ over terminal states

What is the run–time of $C$ on input $\sigma$?

- Behavior of $C$ not entirely determined by $\sigma$
- Probabilistic nature of $C$ influences its run–time
Probabilistic Programs

What does a probabilistic program $C$ do?

- Run program $C$ on initial state $\sigma$
- Obtain final set of (sub-)distributions $\mu$ over terminal states

What is the run–time of $C$ on input $\sigma$?

- Behavior of $C$ not entirely determined by $\sigma$
- Probabilistic nature of $C$ influences its run–time

Better Question:

What is the expected run–time (ERT) of $C$ on input $\sigma$?
Expected Run–Time Phenomena
Expected Run–Time Phenomena

- ERT of $C$ can be finite even if $C$ admits infinite computations
Expected Run–Time Phenomena

- ERT of $C$ can be finite even if $C$ admits infinite computations

\[
x := 1; \text{ while } (1/2) \{ x := 2 \cdot x \}
\]
Expected Run–Time Phenomena

- ERT of $C$ can be finite even if $C$ admits infinite computations

\[ x := 1; \text{while } \left( \frac{1}{2} \right) \{ x := 2 \cdot x \} \]

Program Runtime

\[
\begin{align*}
\text{Run–Time} & \\
\text{Prob.} & \\
\frac{1}{2} & \quad \frac{1}{4} & \quad \frac{1}{8} & \quad \frac{1}{16} & \quad \frac{1}{32}
\end{align*}
\]
Expected Run–Time Phenomena

- ERT of $C$ can be finite even if $C$ admits infinite computations
- Positive almost–sure termination:

\[
x := 1; \text{ while } (1/2) \{ x := 2 \cdot x \}
\]
Expected Run–Time Phenomena

- ERT of $C$ can be finite even if $C$ admits infinite computations
- Positive almost–sure termination:
  - ERT of $C$ is finite

$x := 1; \text{while } (1/2) \{ x := 2 \cdot x \}$
Expected Run–Time Phenomena

- ERT of $C$ can be finite even if $C$ admits infinite computations
- Positive almost–sure termination:
  - ERT of $C$ is finite

---

$x := 1; \text{while } (1/2) \{ x := 2 \cdot x \}; \text{while } (x > 0) \{ x := x - 1 \}$
Expected Run–Time Phenomena

- ERT of $C$ can be finite even if $C$ admits infinite computations
- Positive almost–sure termination:
  - ERT of $C$ is finite
  - Positively almost–surely terminating programs are not closed under sequential composition

---

$x := 1; \text{while } (1/2) \{x := 2 \cdot x\};$

while $(x > 0) \{x := x - 1\}$
Expected Run–Time Phenomena

- ERT of $C$ can be finite even if $C$ admits infinite computations
- Positive almost–sure termination:
  - ERT of $C$ is finite
  - Positively almost–surely terminating programs are not closed under sequential composition
  - Reasoning about positive almost–sure termination is computationally very difficult:

\[
\begin{align*}
  x & := 1; \text{ while } (1/2) \{ x := 2 \cdot x \}; \\
  \text{while } (x > 0) \{ x := x - 1 \}
\end{align*}
\]
Expected Run–Time Phenomena

- ERT of $C$ can be finite even if $C$ admits infinite computations
- Positive almost–sure termination:
  - ERT of $C$ is finite
  - Positively almost–surely terminating programs are not closed under sequential composition
  - Reasoning about positive almost–sure termination is computationally very difficult:

  Strictly more difficult than the termination problem for non–probabilistic programs [MFCS 2015]

---

```plaintext
x := 1; while (1/2) {x := 2 * x};
while (x > 0) {x := x - 1}
```
Expected Run–Time Phenomena

- ERT of $C$ can be finite even if $C$ admits infinite computations
- Positive almost–sure termination:
  - ERT of $C$ is finite
  - Positively almost–surely terminating programs are not closed under sequential composition
  - Reasoning about positive almost–sure termination is computationally very difficult:
    
    Strictly more difficult than the termination problem for non–probabilistic programs [MFCS 2015]

- ERT of $C$ can be infinite, even if $C$ terminates almost–surely\(^1\)

\[
\begin{align*}
  x &:= 1; \text{ while } (\frac{1}{2}) \{x := 2 \cdot x\}; \\
  \text{ while } (x > 0) \{x := x - 1\}
\end{align*}
\]

\(^1\)i.e. with probability 1
Expected Run–Times
Expected Run–Times

ERT if $C$ terminates almost–surely on $\sigma$:

$$\sum_{i=1}^{\infty} i \cdot \Pr(\text{"$C$ terminates after $i$ steps on input $\sigma$"})$$

ERT if $C$ does not terminate almost–surely on $\sigma$:

In general: ERT of $C$ is a function $t: \Sigma \rightarrow \mathbb{R}$

$\sum_{i=1}^{\infty} i \cdot \Pr(\text{"$C$ terminates after $i$ steps on input $\sigma$"})$
Expected Run–Times

- ERT if $C$ terminates almost–surely on $\sigma$:

$$\sum_{i=1}^{\infty} i \cdot \Pr\left( \text{“$C$ terminates after $i$ steps on input $\sigma$”} \right)$$

- ERT if $C$ does not terminate almost–surely on $\sigma$:

$$\infty$$
Expected Run–Times

- ERT if $C$ terminates almost–surely on $\sigma$:

$$\sum_{i=1}^{\infty} i \cdot \Pr\left( \text{"$C$ terminates after } i \text{ steps on input } \sigma\" \right)$$

- ERT if $C$ does not terminate almost–surely on $\sigma$:

$$\infty$$

- In general: ERT of $C$ is a function

$$t: \Sigma \rightarrow \mathbb{R}_{\geq 0}$$
Expected Run–Times

- ERT if $C$ terminates almost–surely on $\sigma$:

  \[ \sum_{i=1}^{\infty} i \cdot \Pr(\text{"$C$ terminates after } i \text{ steps on input } \sigma\text{"}) \]

- ERT if $C$ does not terminate almost–surely on $\sigma$:

  $\infty$

- In general: ERT of $C$ is a function

  \[ t : \Sigma \to \mathbb{R}_{\geq 0}^\infty \]

- Call such a $t$ a run–time.
Expected Run–Times

- ERT if $C$ terminates almost–surely on $\sigma$:
  \[
  \sum_{i=1}^{\infty} i \cdot \Pr\left( \text{"$C$ terminates after } i \text{ steps on input } \sigma \" \right)
  \]

- ERT if $C$ does not terminate almost–surely on $\sigma$:
  \[
  \infty
  \]

- In general: ERT of $C$ is a function
  \[
  t : \Sigma \rightarrow \mathbb{R}_{\geq 0}^\infty
  \]

- Call such a $t$ a run–time. Denote set of run–times by $T$. 
Expected Run–Times

- ERT if $C$ terminates almost–surely on $\sigma$:
  \[ \sum_{i=1}^{\infty} i \cdot \Pr \left( \text{"$C$ terminates after $i$ steps on input $\sigma$"} \right) \]

- ERT if $C$ does not terminate almost–surely on $\sigma$:
  \[ \infty \]

- In general: ERT of $C$ is a function
  \[ t : \Sigma \rightarrow \mathbb{R}_{\geq 0}^\infty \]

- Call such a $t$ a run–time. Denote set of run–times by $\mathbb{T}$.

- Complete partial order on $\mathbb{T}$:
  \[ t_1 \preceq t_2 \text{ iff } \forall \sigma \in \Sigma : t_1(\sigma) \leq t_2(\sigma) \]
Weakest Precondition Reasoning for Expected Run–Times

The ert Transformer

Use a continuation passing style ERT transformer $ert_C$:

$$ert_C(t)$$

Time needed after executing $C$

Expected time needed before executing $C$

ERT in Terms of ert

$$ert_C(0)(σ) = \text{ERT of } C \text{ on input } σ$$
Weakest Precondition Reasoning for Expected Run–Times

The ert Transformer

Use a continuation passing style ERT transformer $ert[C] : T \rightarrow T$. 
The ert Transformer

Use a continuation passing style ERT transformer $\text{ert}[C] : \mathbb{T} \rightarrow \mathbb{T}$.
Weakest Precondition Reasoning for Expected Run–Times

The ert Transformer

Use a continuation passing style ERT transformer $\text{ert}[C] : \mathbb{T} \rightarrow \mathbb{T}$.

\[ C \quad t \]

| time needed after executing $C$ |
The ert Transformer

Use a continuation passing style ERT transformer \( ert[C] : T \rightarrow T \).
Use a continuation passing style ERT transformer $\text{ert}[C]: \mathbb{T} \rightarrow \mathbb{T}$.

\[ \text{ert}[C] (t) \quad C \quad t \]

- expected time needed before executing $C$
- time needed after executing $C$
The ert Transformer

Use a continuation passing style ERT transformer $\text{ert}[C] : \mathbb{T} \rightarrow \mathbb{T}$.

\[
\text{ert}[C](t) \quad C \quad t
\]

- Expected time needed before executing $C$
- Time needed after executing $C$

ERT in Terms of ert

\[
\text{ert}[C](0)(\sigma) = \text{"ERT of } C \text{ on input } \sigma"
\]
Rules for the ert Transformer

\[
\begin{array}{ll}
C & \text{ert} [C] (t) \\
\hline
\text{skip} & 1 + t \\
\end{array}
\]
### Rules for the ert Transformer

<table>
<thead>
<tr>
<th>$C$</th>
<th>ert $[C] (t)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip</td>
<td>1 + $t$</td>
</tr>
<tr>
<td>$x := E$</td>
<td>1 + $t [x/E]$</td>
</tr>
</tbody>
</table>

### Rules for the ert Transformer

<table>
<thead>
<tr>
<th>$C$</th>
<th>ert $[C] (t)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip</td>
<td>$1 + t$</td>
</tr>
<tr>
<td>$x := E$</td>
<td>$1 + t \left[ x/E \right]$</td>
</tr>
</tbody>
</table>

Here, $\left[ x/E \right]$ denotes the result of substituting $x$ with $E$ in the expression.
## Rules for the ert Transformer

<table>
<thead>
<tr>
<th>$C$</th>
<th>ert $[C]$ ($t$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip</td>
<td>1 + $t$</td>
</tr>
<tr>
<td>$x := E$</td>
<td>1 + $t[x/E]$</td>
</tr>
<tr>
<td>$C_1; C_2$</td>
<td>ert $[C_1]$ (ert $[C_2]$ ($t$))</td>
</tr>
</tbody>
</table>
### Rules for the ert Transformer

<table>
<thead>
<tr>
<th>$C$</th>
<th>$\text{ert} \ [C] \ (t)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip</td>
<td>$1 + t$</td>
</tr>
<tr>
<td>$x := E$</td>
<td>$1 + t \ [x/E]$</td>
</tr>
<tr>
<td>$C_1 ; C_2$</td>
<td>$\text{ert} \ [C_1] \ (\text{ert} \ [C_2] \ (t))$</td>
</tr>
<tr>
<td>${C_1} \ □ \ {C_2}$</td>
<td>$\max{\text{ert} \ [C_1] \ (t), \text{ert} \ [C_2] \ (t)}$</td>
</tr>
</tbody>
</table>
## Rules for the ert Transformer

<table>
<thead>
<tr>
<th>C</th>
<th>ert [C] (t)</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip</td>
<td>1 + t</td>
</tr>
<tr>
<td>(x := E)</td>
<td>1 + t [x/E]</td>
</tr>
<tr>
<td>(C_1 ; C_2)</td>
<td>ert [C_1] (ert [C_2] (t))</td>
</tr>
<tr>
<td>{C_1} □ {C_2}</td>
<td>max{ert [C_1] (t), ert [C_2] (t)}</td>
</tr>
<tr>
<td>if ((\xi)) {C_1} else {C_2}</td>
<td>1 + ([\xi: \text{true}] \cdot ert [C_1] (t) + [\xi: \text{false}] \cdot ert [C_2] (t))</td>
</tr>
</tbody>
</table>
# Rules for the ert Transformer

<table>
<thead>
<tr>
<th>Rule</th>
<th>ert ([C] (t))</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip</td>
<td>(1 + t)</td>
</tr>
<tr>
<td>(x := E)</td>
<td>(1 + t \left[ x/E \right])</td>
</tr>
<tr>
<td>(C_1; C_2)</td>
<td>ert ([C_1] (\text{ert} [C_2] (t)))</td>
</tr>
<tr>
<td>({C_1} □ {C_2})</td>
<td>max{ert ([C_1] (t)), ert ([C_2] (t)}}</td>
</tr>
<tr>
<td>if ((\xi) {C_1} \text{ else } {C_2})</td>
<td>(1 + \left[\xi : \text{true}\right] \cdot \text{ert} [C_1] (t) + \left[\xi : \text{false}\right] \cdot \text{ert} [C_2] (t))</td>
</tr>
<tr>
<td>while ((\xi) {C'})</td>
<td>(\text{lfp } X \cdot 1 + \left[\xi : \text{false}\right] \cdot t + \left[\xi : \text{true}\right] \cdot \text{ert} [C'] (X))</td>
</tr>
</tbody>
</table>
Upper Bounds for ert of Loops
Upper Bounds for ert of Loops

Recall the definition of $\text{ert}[\text{while } (\xi) \{C\}] (t)$:

$$\text{lfp } X \cdot 1 + \lfloor \xi : \text{false} \rfloor \cdot t + \lfloor \xi : \text{true} \rfloor \cdot \text{ert } [C] (X)$$
Upper Bounds for ert of Loops

Recall the definition of $\text{ert}[\text{while } (\xi) \{C\}](t)$:

$$\text{lfp } X \cdot 1 + [\xi: \text{false}] \cdot t + [\xi: \text{true}] \cdot \text{ert}[C](X) \tag*{=: F(X)}$$
Upper Bounds for ert of Loops

Recall the definition of $\text{ert}[\text{while } (\xi) \{C\}] (t)$:

$$\text{lfp } X \bullet 1 + [\xi: \text{false}] \cdot t + [\xi: \text{true}] \cdot \text{ert } [C] (X)$$

$$=: F(X)$$

Theorem: Upper Bounds from Upper Invariants
Upper Bounds for ert of Loops

Recall the definition of $\text{ert} \left[ \text{while} \ (\xi) \ {\{C}\}} \right] (t)$:

$$\text{lfp} \ X \cdot 1 + \left[\xi : \text{false}\right] \cdot t + \left[\xi : \text{true}\right] \cdot \text{ert} \ [C] (X) =: F(X)$$

**Theorem: Upper Bounds from Upper Invariants**

If $I \in \mathbb{T}$ is an upper invariant of $\text{while} \ (\xi) \ {\{C}\}$, i.e. if

$$F(I) \leq I$$
Upper Bounds for \( \text{ert} \) of Loops

Recall the definition of \( \text{ert} \left[ \text{while} \left( \xi \right) \{ C \} \right] (t) \): 

\[
\text{lfp } X \cdot 1 + [\xi : \text{false}] \cdot t + [\xi : \text{true}] \cdot \text{ert} \left[ C \right] (X) \\
= : F(X)
\]

**Theorem: Upper Bounds from Upper Invariants**

If \( I \in \mathbb{T} \) is an upper invariant of \( \text{while} \left( \xi \right) \{ C \} \), i.e. if

\[
F(I) \leq I
\]

then

\[
\text{ert} \left[ \text{while} \left( \xi \right) \{ C \} \right] (t) \leq I.
\]
Lower Bounds for ert of Loops
Lower Bounds for ert of Loops

Reasoning on lower bounds is more involved:

Find an argument for being below a least fixed point
Lower Bounds for ert of Loops

Reasoning on lower bounds is more involved:

Find an argument for being below a least fixed point

Theorem: Lower Bounds from Lower $\omega$–Invariants
Lower Bounds for ert of Loops

Reasoning on lower bounds is more involved:

Find an argument for being below a least fixed point

Theorem: Lower Bounds from Lower ω–Invariants

If \( \{I_n\}_{n \in \mathbb{N}} \subseteq T \) is a lower ω–invariant, i.e. if

\[
I_0 \preceq F(0), \quad \text{and} \\
I_{n+1} \preceq F(I_n)
\]
Lower Bounds for ert of Loops

Reasoning on lower bounds is more involved:

Find an argument for being below a least fixed point

Theorem: Lower Bounds from Lower $\omega$–Invariants

If $\{I_n\}_{n \in \mathbb{N}} \subseteq T$ is a lower $\omega$–invariant, i.e. if

$$I_0 \preceq F(0), \quad \text{and}$$

$$I_{n+1} \preceq F(I_n)$$

then

$$\sup_{n \in \mathbb{N}} I_n \preceq \text{ert} [\text{while } (\xi) \{C\} ](t) .$$
Theorem: Completeness of Proof Rules

The presented proof rules are complete
Theorem: Completeness of Proof Rules

The presented proof rules are complete, since $I = \text{lfp } F$ is an upper invariant.
Theorem: Completeness of Proof Rules

The presented proof rules are complete, since $I = \text{lfp } F$ is an upper invariant and a lower $\omega$–invariant is given by

$$I_n = \underbrace{F \circ \cdots \circ F(0)}_{n \text{ times}}.$$
Theorem: Completeness of Proof Rules

The presented proof rules are complete, since $I = \text{lfp } F$ is an upper invariant and a lower $\omega$–invariant is given by

$$I_n = \underbrace{F \circ \cdots \circ F(0)}_{n \text{ times}}.$$ 

Theorem: Bound Refinement

If $I$ is an upper bound and $F(I) \leq I$, then $F(I)$ is also an upper bound.
Theorem: Completeness of Proof Rules

The presented proof rules are complete, since \( I = \text{lfp} F \) is an upper invariant and a lower \( \omega \)-invariant is given by

\[
I_n = F \circ \cdots \circ F(0) \quad \text{\( n \) times}
\]

Theorem: Bound Refinement

If \( I \) is an upper bound and \( F(I) \leq I \), then \( F(I) \) is also an upper bound. Dually for lower bounds.
Is the ert Calculus a Reasonable Run–Time Model?

Correspondence to an operational semantics: ert coincides with expected reward in the operational MDP a la [QEST 2012] and [MFPS 2015]. Enables bounded model checking of expected run–times.

Nielson's Hoare–style logic for reasoning about run–time orders of magnitude of deterministic programs:

ert is sound and complete with respect to Nielson's logic. ert calculus is arguably easier to apply — no additional logical variables!
Is the ert Calculus a Reasonable Run–Time Model?

- Correspondence to an operational semantics:
Is the ert Calculus a Reasonable Run–Time Model?

- Correspondence to an operational semantics:
  - Operational model defined in terms of a reward MDP à la [QEST 2012] and [MFPS 2015]
Is the ert Calculus a Reasonable Run–Time Model?

- Correspondence to an operational semantics:
  - Operational model defined in terms of a reward MDP à la [QEST 2012] and [MFPS 2015]
  - ert coincides with expected reward in the operational MDP
Is the ert Calculus a Reasonable Run–Time Model?

- Correspondence to an operational semantics:
  - Operational model defined in terms of a reward MDP à la [QEST 2012] and [MFPS 2015]
  - ert coincides with expected reward in the operational MDP
  - Enables bounded model checking of expected run–times
Is the ert Calculus a Reasonable Run–Time Model?

- Correspondence to an operational semantics:
  - Operational model defined in terms of a reward MDP à la [QEST 2012] and [MFPS 2015]
  - ert coincides with expected reward in the operational MDP
  - Enables bounded model checking of expected run–times

- Nielson’s Hoare–style logic for reasoning about run–time orders of magnitude of deterministic programs:
Is the ert Calculus a Reasonable Run–Time Model?

- Correspondence to an operational semantics:
  - Operational model defined in terms of a reward MDP à la [QEST 2012] and [MFPS 2015]
  - ert coincides with expected reward in the operational MDP
  - Enables bounded model checking of expected run–times

- Nielson’s Hoare–style logic for reasoning about run–time orders of magnitude of deterministic programs:
  - Nielson’s logic relies on introducing additional logical variables
Is the ert Calculus a Reasonable Run–Time Model?

- Correspondence to an operational semantics:
  - Operational model defined in terms of a reward MDP à la [QEST 2012] and [MFPS 2015]
  - ert coincides with expected reward in the operational MDP
  - Enables bounded model checking of expected run–times

- Nielson’s Hoare–style logic for reasoning about run–time orders of magnitude of deterministic programs:
  - Nielson’s logic relies on introducing additional logical variables
  - ert is sound and complete with respect to Nielson’s logic
Is the ert Calculus a Reasonable Run–Time Model?

- Correspondence to an operational semantics:
  - Operational model defined in terms of a reward MDP à la [QEST 2012] and [MFPS 2015]
  - ert coincides with expected reward in the operational MDP
  - Enables bounded model checking of expected run–times

- Nielson’s Hoare–style logic for reasoning about run–time orders of magnitude of deterministic programs:
  - Nielson’s logic relies on introducing additional logical variables
  - ert is sound and complete with respect to Nielson’s logic
  - ert calculus is arguably easier to apply — no additional variables!
Case Study: The Coupon Collector’s Problem

The coupon collector is a well-known problem. We model it by the following algorithm:

```plaintext
CP := [0, ..., 0];
i := 1;
x := N;
while (x > 0) {
    while (CP[i] ≠ 0) {
        i := Unif[1...N];
    }
    CP[i] := 1;
    x := x - 1;
}
```

Using ert, we can analyze the ERT of the above algorithm directly on the source code given above:

\[
\text{ert}\[\text{coupon\_coll}](0) = 4 + (N > 0 \cdot 2^N \cdot (2 + H_{N-1}))
\]

Harmonic number \(H_N-1\) is in \(Θ(\log N)\).

Coupon collector program runs in \(Θ(N \cdot \log N)\) for \(N > 0\).
Case Study: The Coupon Collector’s Problem

- The coupon collector is a well-known problem
Case Study: The Coupon Collector's Problem

The coupon collector is a well-known problem.

The coupon collector is a well-known problem. We model it by the following algorithm:

1. Let \( cp := \[0, \ldots, 0\] \) and \( i := 1 \), \( x := N \).
2. While \( x > 0 \):
   a. While \( cp[i] \neq 0 \):
      i. \( i := \approx \text{Unif}[1..N] \) (uniform distribution between 1 and N);
   b. \( cp[i] := 1 \);
   c. \( x := x - 1 \) (decrement x by 1).

Using \( \text{ert} \), we can analyze the ERT of the above algorithm directly on the source code given above:

\[
\text{ert} \left[ \text{coup.coll.} \right] (0) = 4 + N \cdot 2^N \cdot \left( 2 + H_N - 1 \right)
\]

Harmonic number \( H_N - 1 \) is in \( \Theta(\log N) \).

The coupon collector program runs in \( \Theta(N \cdot \log N) \) for \( N > 0 \).
Case Study: The Coupon Collector’s Problem

The coupon collector is a well-known problem. We model it by the following algorithm:

\[ \begin{aligned}
& \text{cp} := [0, \ldots, 0] ; \\
& i := 1 ; \\
& x := N ; \\
& \text{while } (x > 0) \{ \\
& \quad \text{while } (\text{cp}[i] \neq 0) \{ \\
& \quad \quad i := \approx \text{Unif}[1 \ldots N] ; \\
& \quad \} ; \\
& \quad \text{cp}[i] := 1 ; \\
& \quad x := x - 1 \\
& \} 
\end{aligned} \]

Using ert, we can analyze the ERT of the above algorithm directly on the source code given above:

\[ \text{ERT}(\text{coup.coll.}) (0) = 4 + \left( N > 0 \cdot 2^N \cdot (2 + H_N - 1) \right) \]

Harmonic number \( H_N - 1 \) is in \( \Theta(\log N) \)

Coupon collector program runs in \( \Theta(N \cdot \log N) \) for \( N > 0 \).
Case Study: The Coupon Collector’s Problem

- The coupon collector is a well–known problem
Case Study: The Coupon Collector’s Problem

- The coupon collector is a well–known problem
- We model it by the following algorithm:

\[
\begin{align*}
  cp &:= [0, \ldots, 0]; i := 1; x := N; \\
  \text{while} (x > 0) \{ \\
    \text{while} (cp[i] \neq 0) \{ i \approx \text{Unif}[1\ldots N] \}; \\
    cp[i] := 1; x := x - 1 \}
\end{align*}
\]
Case Study: The Coupon Collector’s Problem

- The coupon collector is a well-known problem
- We model it by the following algorithm:

\[
\begin{align*}
cp & := [0, \ldots, 0]; i := 1; x := N; \\
\text{while}(x > 0) \{ \\
& \quad \text{while}(cp[i] \neq 0) \{ i \approx \text{Unif}[1 \ldots N] \}; \\
& \quad cp[i] := 1; x := x - 1 \}
\end{align*}
\]

- Using ert, we can analyze the ERT of the above algorithm directly on the source code given above:
Case Study: The Coupon Collector’s Problem

- The coupon collector is a well-known problem.

- We model it by the following algorithm:

  \[ cp := [0, \ldots, 0]; \ i := 1; \ x := N; \]
  \[ \text{while} (x > 0) \{ \]
  \[ \quad \text{while} (cp[i] \neq 0) \{ \ i := \text{Unif}[1 \ldots N] \}; \]
  \[ \quad cp[i] := 1; \ x := x - 1 \} \]

- Using ert, we can analyze the ERT of the above algorithm directly on the source code given above:

  \[ \text{ert} [\text{coup. coll.}] (0) = 4 + [N > 0] \cdot 2N \cdot (2 + \mathcal{H}_{N-1}) \]
Case Study: The Coupon Collector’s Problem

- The coupon collector is a well–known problem
- We model it by the following algorithm:
  \[
  cp := [0, \ldots, 0]; i := 1; x := N; \\
  \text{while } (x > 0) \{ \\
  \quad \text{while } (cp[i] \neq 0) \{ i \approx \text{Unif}[1\ldots N] \}; \\
  \quad cp[i] := 1; x := x - 1 \} \\
  \]
- Using ert, we can analyze the ERT of the above algorithm directly on the source code given above:
  \[
  \text{ert}[coup.\ coll.](0) = 4 + [N > 0] \cdot 2N \cdot (2 + \mathcal{H}_{N-1}) \\
  \]
- Harmonic number $\mathcal{H}_{N-1}$ is in $\Theta(\log N)$
Case Study: The Coupon Collector’s Problem

- The coupon collector is a well–known problem.

- We model it by the following algorithm:

  
  \[
  \begin{align*}
  cp & := [0, \ldots, 0] ; \ i := 1 ; \ x := N ; \\
  \text{while} & \ (x > 0) \ \{ \\
  & \ \ \ \text{while} \ (cp[i] \neq 0) \ \{ \ i := \text{Unif}[1 \ldots N] \ \}; \\
  & \ \ \ cp[i] := 1 ; \ x := x - 1 \ \\
  \} 
  \end{align*}
  \]

- Using ert, we can analyze the ERT of the above algorithm directly on the source code given above:

  \[
  \text{ert} \ [coup. \ coll.] \ (0) = 4 + [N > 0] \cdot 2N \cdot (2 + H_{N-1})
  \]

- Harmonic number \( H_{N-1} \) is in \( \Theta(\log N) \).

- Coupon collector program runs in \( \Theta(N \cdot \log N) \) for \( N > 0 \).
Summary

ert is an easy to understand weakest–precondition–style calculus for reasoning about ERT of probabilistic programs.

er is sound and complete for reasoning about expected run–times and positive almost–sure termination.

er comes with proof rules for reasoning about loops.

er is a powerful alternative to ranking super–martingales.

er is applicable to tricky real–world examples which are difficult to reason about by formal verification techniques.

ert is Isabelle/HOL certified (courtesy of Johannes Hötzl, TUM).

Future work: recursion, conditioning, run–time variance.

Thank you for your kind attention!
Summary

- ert is an easy to understand weakest–precondition–style calculus for reasoning about ERT of probabilistic programs.
Summary

- ert is an easy to understand weakest–precondition–style calculus for reasoning about ERT of probabilistic programs

- ert is sound and complete for reasoning about expected run–times and positive almost–sure termination
Summary

- **ert** is an **easy to understand weakest–precondition–style calculus** for reasoning about ERT of probabilistic programs.
- **ert** is **sound and complete** for reasoning about expected run–times and positive almost–sure termination.
- **ert** comes with **proof rules** for reasoning about loops.
Summary

- ert is an easy to understand weakest–precondition–style calculus for reasoning about ERT of probabilistic programs
- ert is sound and complete for reasoning about expected run–times and positive almost–sure termination
- ert comes with proof rules for reasoning about loops
- ert is a powerful alternative to ranking super–martingales
Summary

- Ert is an easy to understand weakest-precondition-style calculus for reasoning about ERT of probabilistic programs.
- Ert is sound and complete for reasoning about expected run-times and positive almost-sure termination.
- Ert comes with proof rules for reasoning about loops.
- Ert is a powerful alternative to ranking super-martingales.
- Ert is applicable to tricky real-world examples which are difficult to reason about by formal verification techniques.
Summary

- eRT is an easy to understand weakest–precondition–style calculus for reasoning about ERT of probabilistic programs.
- eRT is sound and complete for reasoning about expected run–times and positive almost–sure termination.
- eRT comes with proof rules for reasoning about loops.
- eRT is a powerful alternative to ranking super–martingales.
- eRT is applicable to tricky real–world examples which are difficult to reason about by formal verification techniques.

ERT is Isabelle/HOL certified (courtesy of Johannes Hölzl, TUM).
Summary

- ert is an easy to understand weakest–precondition–style calculus for reasoning about ERT of probabilistic programs
- ert is sound and complete for reasoning about expected run–times and positive almost–sure termination
- ert comes with proof rules for reasoning about loops
- ert is a powerful alternative to ranking super–martingales
- ert is applicable to tricky real–world examples which are difficult to reason about by formal verification techniques

ert is Isabelle/HOL certified (courtesy of Johannes Hölzl, TUM)

Future work: recursion, conditioning, run–time variance
Summary

- Ert is an easy to understand weakest–precondition–style calculus for reasoning about ERT of probabilistic programs.
- Ert is sound and complete for reasoning about expected run–times and positive almost–sure termination.
- Ert comes with proof rules for reasoning about loops.
- Ert is a powerful alternative to ranking super–martingales.
- Ert is applicable to tricky real–world examples which are difficult to reason about by formal verification techniques.
- Ert is Isabelle/HOL certified (courtesy of Johannes Hölzl, TUM).
- Future work: recursion, conditioning, run–time variance.

Thank you for your kind attention!
Backup Slides: The Actual Rule for Assignments

\[ C \quad \text{ert} [C] (t) \]

\[ x \approx \mu \quad 1 + \lambda \sigma \cdot E_{[\mu]} (\sigma) (\lambda v. t [x/v] (\sigma)) \]
Backup Slides: ert Calculations and Proof Rule Application

Example 4 (Geometric distribution). Consider loop

\[ C_{geo} : \text{while (c = 1) \{ c :\approx \frac{1}{2} \cdot \langle 0 \rangle + \frac{1}{2} \cdot \langle 1 \rangle \}}. \]

From the calculations below we conclude that \( I = 1 + [c = 1] \cdot 4 \) is an upper invariant with respect to 0:

\[
1 + [c \neq 1] \cdot 0 + [c = 1] \cdot \text{ert} [c :\approx \frac{1}{2} \cdot \langle 0 \rangle + \frac{1}{2} \cdot \langle 1 \rangle] (I)
\]

\[
= 1 + [c = 1] \cdot (1 + \frac{1}{2} \cdot I [c/0] + \frac{1}{2} \cdot I [c/1])
\]

\[
= 1 + [c = 1] \cdot (1 + \frac{1}{2} \cdot (1 + [0 = 1] \cdot 4) + \frac{1}{2} \cdot (1 + [1 = 1] \cdot 4) = 1 + [c = 1] \cdot 4 = I \leq I
\]

Then applying Theorem 3 we obtain

\[
\text{ert} [C_{geo}] (0) \leq 1 + [c = 1] \cdot 4.
\]

In words, the expected run–time of \( C_{geo} \) is at most 5 from any initial state where \( c = 1 \) and at most 1 from the remaining states. \( \triangle \)
Backup Slides: Operational RMDP

\[ C_{\text{trunc}} : \text{if} \ (\frac{1}{2} \cdot \langle \text{true} \rangle + \frac{1}{2} \cdot \langle \text{false} \rangle) \{ \text{succ} := \text{true} \} \text{ else } \{
\text{if} \ (\frac{1}{2} \cdot \langle \text{true} \rangle + \frac{1}{2} \cdot \langle \text{false} \rangle) \{ \text{succ} := \text{true} \}
\text{ else } \{ \text{succ} := \text{false} \}\}
\]
Backup Slides: Park’s Lemma
Backup Slides: Park’s Lemma

\[
F(I) \leq I \text{ implies } \text{lfp } F \leq I
\]

\[
\text{gfp } F
\]

\[
\text{lfp } F
\]

\[
\infty
\]

\[
I \rightarrow 0
\]

Backup Slides: Park’s Lemma
Backup Slides: Park’s Lemma

\[ F(I) \leq I \quad \text{implies} \quad \text{lfp } F \leq I \]
Backup Slides: Park’s Lemma

\[ F(I) \leq I \implies \text{lfp } F \leq I \]
Backup Slides: Park’s Lemma

\[ F(I) \leq I \implies \text{lfp } F \leq I \]