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
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
Proabilistic 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

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

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

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

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

Probabilistic Programs

Syntax of Probabilistic Programs

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

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

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

What is probabilistic about that language?
What is probabilistic about that language?

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

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

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
### Syntax of Probabilistic Programs

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

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

- E.g. \(\frac{2}{3}\langle\text{true}\rangle + \frac{1}{3}\langle\text{false}\rangle\)
Syntax of Probabilistic Programs

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

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

\begin{verbatim}
\texttt{x := 1; while (1/2) \{x := 2 \cdot x\}}
\end{verbatim}
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
- 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

```plaintext
x := 1; 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:

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

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

- ERT of $C$ can be **infinite**, even if $C$ terminates almost–surely

\[ x := 1; \text{while } (1/2) \{ x := 2 \cdot x \}; \text{while } (x > 0) \{ x := x - 1 \} \]

\[ ^1 \text{i.e. with probability } 1 \]
Expected Run–Times

Motivation

Expected Run–Times

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

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

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

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

Complete partial order on $T$:

$t_1 \preceq t_2$ if $\forall \sigma \in \Sigma: t_1(\sigma) \leq t_2(\sigma)$.

Denote set of run–times by $T$.

Kaminski, Katoen, Matheja, Olmedo

Weakest Precondition Reasoning for Expected Run–Times

4.4.2016
Expected Run–Times

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

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

$\infty$
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

\[ t : \Sigma \rightarrow \mathbb{R}_{\geq0}^{\infty} \]
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

$$t : \Sigma \rightarrow \mathbb{R}_{\geq 0}$$

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

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

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

- 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 $\mathbb{T}$. 
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

$$t : \Sigma \to \mathbb{R}_{\geq 0}$$

■ Call such a $t$ a run–time. Denote set of run–times by $T$.

■ Complete partial order on $T$:

$$t_1 \preceq t_2 \iff \forall \sigma \in \Sigma : t_1(\sigma) \leq t_2(\sigma)$$

Kaminski, Katoen, Matheja, Olmedo

Weakest Precondition Reasoning for Expected Run–Times

The ert Transformer

<table>
<thead>
<tr>
<th>Use a continuation passing style ERT transformer | ( ert )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time needed after executing ( C )</td>
</tr>
</tbody>
</table>

Kaminski, Katoen, Matheja, Olmedo

Weakest Precondition Reasoning for Expected Run–Times
Weakest Precondition Reasoning for Expected Run–Times

The ert Transformer

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

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

$C$
The ert Transformer

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

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

- \text{expected time needed before executing } C
- \text{time needed after executing } C
The ert Transformer

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

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

C \quad \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

<table>
<thead>
<tr>
<th>$C$</th>
<th>ert $[C]$ $(t)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip</td>
<td>$1 + t$</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>
</tbody>
</table>
### Rules for the ert Transformer

<table>
<thead>
<tr>
<th>C</th>
<th>ert (<a href="t">C</a>)</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>$1 + t[x/E]$</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>ert $[C] (t)$</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>skip</code></td>
<td>$1 + t$</td>
</tr>
<tr>
<td>$x := E$</td>
<td>$1 + t \cdot x/E$</td>
</tr>
<tr>
<td>$C_1 ; C_2$</td>
<td>ert $[C_1] (ert [C_2] (t))$</td>
</tr>
<tr>
<td><code>{C_1} \sqcap {C_2}</code></td>
<td>max{ert $[C_1] (t)$, 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>${C_1} \square {C_2}$</td>
<td>$\max{\text{ert} [C_1] (t), \text{ert} [C_2] (t)}$</td>
</tr>
</tbody>
</table>
| if ($\xi$) $\{C_1\}$ else $\{C_2\}$ | $1 + \llbracket \xi: \text{true} \rrbracket \cdot \text{ert} [C_1] (t)$
| | $+ \llbracket \xi: \text{false} \rrbracket \cdot \text{ert} [C_2] (t)$ |
# 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>${C_1} \ \Box \ {C_2}$</td>
<td>$\max{\text{ert} \ [C_1] \ (t), \text{ert} \ [C_2] \ (t)}$</td>
</tr>
<tr>
<td>if ($\xi$) ${C_1}$ else ${C_2}$</td>
<td>$1 + \llbracket \xi : \text{true} \rrbracket \cdot \text{ert} \ [C_1] \ (t)$ $+ \llbracket \xi : \text{false} \rrbracket \cdot \text{ert} \ [C_2] \ (t)$</td>
</tr>
<tr>
<td>while ($\xi$) ${C'}$</td>
<td>$\text{lfp} X. \ 1 + \llbracket \xi : \text{false} \rrbracket \cdot t$ $+ \llbracket \xi : \text{true} \rrbracket \cdot \text{ert} \ [C'] \ (X)$</td>
</tr>
</tbody>
</table>
Upper Bounds for ert of Loops

Recall the definition of ert:

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

Theorem: Upper Bounds from Upper Invariants

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

\[ \text{ert}[\text{while}(\xi)\{C\}](t) \preceq I. \]
Upper Bounds for ert of Loops

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

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

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

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

$$=: F(X)$$
Upper Bounds for ert of Loops

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

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

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

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

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

**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 ert of Loops

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

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

$$=: F(X)$$

Theorem: Upper Bounds from Upper Invariants

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

$$F(I) \leq I$$

then

$$\text{ert}[\text{while } (\xi) \{C\}] (t) \leq I.$$
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
Weakest Precondition Reasoning for Expected Run–Times

Reasoning about ert

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

\[
I_0 \preceq F(0), \quad \text{and} \quad I_{n+1} \preceq F(I_n)
\]
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

\[
I_0 \preceq F(0), \quad \text{and}\quad I_{n+1} \preceq F(I_n)
\]

then

\[
\sup_{n \in \mathbb{N}} I_n \preceq ert[\text{while } (\xi) \{C\}](t).
\]
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

$$I_n = F \circ \cdots \circ F(0).$$

$n$ 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

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

Theorem: Bound Refinement

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

```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:

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

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. We model it by the following algorithm:

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

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

\[ \text{ERT} [\text{coupon}\_\text{collect\_}\text{or\_}\text{theory}] (0) = 4 + N \cdot 2^N \cdot (2 + H_N - 1) \]

Harmonic number \( H_N \) 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
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:

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

Harmonic number \( H_N \) 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:

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

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

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

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

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

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

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

```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:

```plaintext
ert [coup. coll.] (0) = 4 + [N > 0] · 2N · (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:
  
  \[
  cp := [0, \ldots, 0]; \ i := 1; \ x := N; \\
  \text{while} (x > 0) \{ \\
  \quad \text{while} (cp[i] \neq 0) \{ \ i := \text{Unif}[1 \ldots N] \}; \\
  \quad cp[i] := 1; \ x := x - 1 \}
  \]

- Using ert, we can analyze the ERT of the above algorithm directly on the source code given above:
  
  \[
  \text{ert}[\text{coup. coll.}](0) = 4 + [N > 0] \cdot 2N \cdot (2 + \mathcal{H}_{N-1})
  \]

- Harmonic number \( \mathcal{H}_{N-1} \) is in \( \Theta(\log N) \)

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

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

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

\[
\begin{array}{c}
C \\
\text{ert} \ [C] (t)
\end{array}
\]

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

**Example 4 (Geometric distribution).** Consider loop

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

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

\[
\begin{align*}
1 + [c \neq 1] \cdot 0 &+ [c = 1] \cdot \text{ert} [c := 1/2 \cdot \langle 0 \rangle + 1/2 \cdot \langle 1 \rangle ] (I) \\
&= 1 + [c = 1] \cdot (1 + \frac{1}{2} \cdot I [c/0] + \frac{1}{2} \cdot I [c/1]) \\
&= 1 + [c = 1] \cdot (1 + \frac{1}{2} \cdot (1 + [0 = 1] \cdot 4) + \frac{1}{2} \cdot (1 + [1 = 1] \cdot 4)) \\
&= 1 + [c = 1] \cdot 4 = I \preceq I
\end{align*}
\]

Then applying Theorem 3 we obtain

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

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

\[ C_{\text{trunc}} : \quad \text{if} \ (\frac{1}{2} \cdot \langle \text{true} \rangle + \frac{1}{2} \cdot \langle \text{false} \rangle) \ \{ \text{succ} := \text{true} \} \ \text{else} \ \{ \]
\[
\text{if} \ (\frac{1}{2} \cdot \langle \text{true} \rangle + \frac{1}{2} \cdot \langle \text{false} \rangle) \ \{ \text{succ} := \text{true} \}
\]
\[
\text{else} \ \{ \text{succ} := \text{false} \}
\]

\[
\begin{array}{c}
1 \\
\rightarrow \langle C, \sigma \rangle \\
\end{array}
\]

\[
\begin{array}{c}
\frac{1}{2} \\
\frac{1}{2} \\
\frac{1}{2} \\
\end{array}
\]

\[
\begin{array}{cc}
1 \langle \text{succ} := \text{true}, \sigma \rangle \quad \langle C', \sigma \rangle \quad 1 \\
\end{array}
\]

\[
\begin{array}{c}
f(\sigma[\text{succ}/\text{true}]) \\
\langle \bot, \sigma[\text{succ}/\text{true}] \rangle \\
\langle \text{succ} := \text{false}, \sigma \rangle \quad 1 \\
\end{array}
\]

\[
\begin{array}{c}
0 \langle \text{sink} \rangle \quad \langle \bot, \sigma[\text{succ}/\text{false}] \rangle \\
f(\sigma[\text{succ}/\text{false}])
\end{array}
\]
Backup Slides: Park’s Lemma
Backup Slides: Park’s Lemma
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

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

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