Quantitative Separation Logic
– A Logic for Reasoning about Probabilistic Pointer Programs –

Benjamin Kaminski, Joost-Pieter Katoen, Christoph Matheja¹, Thomas Noll

WIAI-Fakultätskolloquium, Universität Bamberg, Germany

November 7, 2019

¹supported by DFG Grant NO 401/2-1
Introduction

Content

Introduction

Separation Logic  [John Reynolds, Peter O’Hearn]

Probabilistic Weakest Preconditions  [Dexter Kozen, Annabelle McIver, Carroll Morgan]

Quantitative Separation Logic

Case Studies

Conclusion
Why Probabilistic Pointer Programs?

Since 1970s: randomised algorithms

- “Randomised skip list algorithms have the same asymptotic expected time bounds as balanced trees and are simpler, faster, and use less space.” [Pugh 1989]
- “The expected running time of randomised splay trees is smaller than deterministic variants” [Albers & Karpinski 2002]
Why Probabilistic Pointer Programs?

Since 1970s: randomised algorithms

- “Randomised skip list algorithms have the same asymptotic expected time bounds as balanced trees and are simpler, faster, and use less space.”  
  [Pugh 1989]
- “The expected running time of randomised splay trees is smaller than deterministic variants”  
  [Albers & Karpinski 2002]

More recently: approximate computing, artificial intelligence, . . .

- Lots of probabilistic programming languages (often without precise semantics)
- Prominent example: Stan (10k+ active users)
What are Probabilistic Programs?

- “Ordinary” programs with the additional ability to flip coins
  
  \{ \text{skip} \} \left[ \frac{1}{3} \right] \{ x := x + 2 \}
What are Probabilistic Programs?

- “Ordinary” programs with the additional ability to flip coins
  \[
  \{ \text{skip} \} \left[ \frac{1}{3} \right] \{ x := x + 2 \} \]

- What does a probabilistic program \( C \) do?
  
  - Run program \( C \) on initial state \( \sigma \)
  
  - Obtain distribution \( \llbracket C \rrbracket_\sigma \) over final states
What are Probabilistic Programs?

- “Ordinary” programs with the additional ability to flip coins
  \[
  \{ \text{skip} \} [\frac{1}{3}] \{ x := x + 2 \}
  \]

- What does a probabilistic program \( C \) do?
  - Run program \( C \) on initial state \( \sigma \)
  - Obtain (sub-)distribution \( \llbracket C \rrbracket_{\sigma} \) over final states
What are Probabilistic Programs?

• “Ordinary” programs with the additional ability to flip coins
  \[
  \{ \text{skip} \} \left[ \frac{1}{3} \right] \{ x := x + 2 \}
  \]

• What does a probabilistic program \( C \) do?
  – Run program \( C \) on initial state \( \sigma \)
  – Obtain (sub-)distribution \( [C]_\sigma \) over final states

• Operational semantics is a Markov chain

\[
\begin{align*}
&\langle \{ \text{skip} \} \left[ \frac{1}{3} \right] \{ x := x + 2 \}, \sigma \rangle \\
\downarrow &\quad 1/3 \\
&\langle \text{skip}, \sigma \rangle \\
\downarrow &\quad 2/3 \\
&\langle x := x + 2, \sigma \rangle \\
\downarrow &\quad 1/3 \\
&\langle \bot, \sigma \rangle \\
\downarrow &\quad 2/3 \\
&\langle \bot, \sigma [x/x + 2] \rangle
\end{align*}
\]
What are **Probabilistic Programs**?

- "Ordinary" programs with the additional ability to flip coins
  \[
  \{ \text{skip} \} \left[ \frac{1}{3} \right] \{ x := x + 2 \}
  \]

- What does a probabilistic program \( C \) do?
  - Run program \( C \) on initial state \( \sigma \)
  - Obtain (sub-)distribution \( [[C]]_\sigma \) over final states

- Operational semantics is a Markov chain

**Questions**

- What is the **probability** that a program behaves correctly?
- What is a program’s **expected behaviour**?
- What is its **expected runtime**?

\[
\langle \{ \text{skip} \} \left[ \frac{1}{3} \right] \{ x := x + 2 \}, \sigma \rangle
\]

\[
\langle \text{skip}, \sigma \rangle \quad \langle x := x + 2, \sigma \rangle
\]

\[
\langle \downarrow, \sigma \rangle \quad \langle \downarrow, \sigma [x/x + 2] \rangle
\]
Randomised Algorithms

Example (Array randomisation)

```plaintext
randomize(array, n) {
    i := 0;
    while (0 ≤ i < n) {
        j := uniform(i, n − 1);
        swap(array, i, j);
        i := i + 1
    }
}
```

Question

Is the probability of each permutation \(\frac{1}{n!}\)?
Randomised Algorithms

Example (Array randomisation)

\textbf{randomize}(array, n)\{
  \hspace{1em} i := 0; \\
  \hspace{1em} \textbf{while}(0 \leq i < n)\{
    \hspace{2em} j := \text{uniform}(i, n - 1); \\
    \hspace{2em} \text{swap}(array, i, j); \\
    \hspace{2em} i := i + 1
  \hspace{1em} } 
}\}

\textbf{Question}

Is the probability of each permutation \(\frac{1}{n!}\)?
Example (Faulty garbage collector)

```c
delete(x : tree) {
    if (x \neq 0) { // fails with probability \( p \)
        { skip } [p] {
            l := x.left; r := x.right;
            delete(l);
            delete(r);
            free(x);
        }
    }
}
```
Example (Faulty garbage collector)

```latex
\begin{verbatim}
delete(x : tree) {
  if (x \neq 0) { // fails with probability \( p \)
    \{ skip \} [p] {
      l := x.left; r := x.right;
      delete(l);
      delete(r);
      free(x);
    }
  }
}
\end{verbatim}
```

Question

What is the probability that on termination the heap is empty?
“In no other branch of mathematics is it so easy to make mistakes as in probability theory.”

\[
x := 1; \text{while}(x = 1)\{\{\text{skip}\} \frac{1}{2} \{x := 0\}\}
\]
Probabilities are Problematic

“In no other branch of mathematics is it so easy to make mistakes as in probability theory.”

[Tijms 2004]

$x := 1; \text{while} (x = 1) \{ \{ \text{skip} \} [1/2] \{ x := 0 \} \}$

- does not always terminate
- terminates almost-surely
- expected runtime: finite
Probabilities are Problematic

“In no other branch of mathematics is it so easy to make mistakes as in probability theory.”

[Tijms 2004]

\[ x := 1; \textbf{while} (x = 1) \{ \{ \textbf{skip} \}^{\frac{1}{2}} \{ x := 0 \} \} \]

- does not always terminate
- terminates \textit{almost-surely}
- expected runtime: finite

\[ x := 1; y := 1; \textbf{while} (x = 1) \{ \{ y := 2 \cdot y \}^{\frac{1}{2}} \{ x := 0 \} \} \]

- does not always terminate
- terminates \textit{almost-surely}
- expected runtime: \textit{finite}
Probabilities are Problematic

“In no other branch of mathematics is it so easy to make mistakes as in probability theory.”

[Tijms 2004]

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

- does not always terminate
- terminates almost-surely
- expected runtime: finite

- does not always terminate
- terminates almost-surely
- expected runtime: infinite
Probabilities are Problematic

“In no other branch of mathematics is it so easy to make mistakes as in probability theory.”

[Tijms 2004]

\[ x := 1; \text{while}(x = 1)\{\{\text{skip}\}[\frac{1}{2}]\{x := 0\}\} \]

- does not always terminate
- terminates almost-surely
- expected runtime: finite

\[ x := 1; y := 1; \text{while}(x = 1)\{\{y := 2 \cdot y\}[\frac{1}{2}]\{x := 0\}\} \]

- does not always terminate
- terminates almost-surely
- expected runtime: finite

Proving that the expected runtime is finite is strictly harder than proving termination!

[Acta Inf. 2018]
Pointers are Problematic

Dereferencing null pointers, pointer aliasing, memory leaks, ...
Goal: Formal Verification for **Probabilistic Pointer Programs**

- **Hoare logic**
- **Probabilistic Weakest Preconditions**
- **Separation Logic**

Thomas Noll
November 7, 2019
Goal: Formal Verification for Probabilistic Pointer Programs

Diagram:

- Hoare logic
- Probabilistic Weakest Preconditions
- Separation Logic
- Quantitative Separation Logic

Thomas Noll
November 7, 2019
1. Introduction

System reliability is a major challenge in the design of emerging architectures. Energy efficiency and circuit scaling are becoming major goals when designing new devices. However, aggressively pursuing these design goals can often increase the frequency of soft errors in small [67] and large systems [30] alike. Researchers have developed numerous techniques for detecting and masking soft errors in both hardware [23] and software [20, 53, 57, 64]. These techniques typically come at the price of increased execution time, increased energy consumption, or both.

Many computations, however, can tolerate occasional unmasked errors. An approximate computation (including many multimedia, financial, machine learning, and big data analytics applications) can often acceptably tolerate occasional errors in its execution and/or the data that it manipulates [16, 44, 59]. A checkable computation can be augmented with an efficient checker that verifies the acceptability of the computation’s results [8, 9, 35, 55]. If the checker does not detect an error, it can reexecute the computation to obtain an acceptable result.

For both approximate and checkable computations, operating without (or with at most selectively applied) mechanisms that detect and mask soft errors can produce 1) fast and energy efficient execution and 2) delivers acceptably accurate results often enough to satisfy the needs of their users despite the presence of unmasked soft errors.

1.1 Background

Researchers have identified a range of both approximate
Skip Lists: A Probabilistic Alternative to Balanced Trees

William Pugh

Abstract

Binary trees can be used for representing abstract data types such as dictionaries and ordered lists. They work well when the elements are inserted in a random order. Some sequences of operations, such as inserting the elements in order, produce degenerate data structures that perform very poorly. If it were possible to randomly permute the list to be inserted, trees would work well with high probability for any input sequence. In most cases queries must be answered online, so rearranging the input is impractical. Balanced tree algorithms maintain the tree as operations are performed to maintain certain balance conditions and ensure good performance.

Skip lists are a probabilistic alternative to balanced trees. Skip lists are balanced by consulting a random number generator. Although skip lists have bad worst-case performance, most sequences consistently generate the worst-case performance (much like quicksort when the pivot element is chosen randomly). It is very unlikely a skip list data structure will be significantly unbalanced (e.g., for a dictionary of more than 2^38 elements, the chance that a search will take more than three times the expected time is less than one in a million). Skip lists have balance properties similar to self-adjusting tree algorithms. Skip lists are also very space efficient. They can easily be configured to require an average of 1 log base 2 of n pointers per element (or even less) and do not require balance or priority information to be stored with each node.

1. Introduction

Many computations, however, can tolerate occasional soft errors. Full detection and masking of soft errors is challenging, expensive and, for some applications, unnecessary. For example, approximate computing applications (such as multimedia processing, machine learning, and big data analytics) can often naturally tolerate soft errors.

We present Rely, a programming language that enables developers to reason about the quantitative reliability of an application—namely, the probability that it produces the correct result when executed on unreliable hardware. Rely allows developers to specify the reliability requirements for each value that a function produces.

We present a static quantitative reliability analysis that verifies quantitative requirements on the reliability of an application, enabling a developer to perform sound and verifiable reliability engineering. The analysis takes a Rely program with a reliability specification and a hardware specification that characterizes the reliability of the underlying hardware components and verifies that the program satisfies its reliability specification when executed on the underlying reliable hardware platform. We demonstrate the application of quantitative reliability analysis on six computations implemented in Rely.

Categories and Subject Descriptors F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs
Abstract

Emerging high-performance architectures are anticipated to contain unreliable components that may exhibit soft errors which silently corrupt the results of computations. Full detection and masking of soft errors is challenging, expensive and, for some applications, unnecessary. For example, approximate computing applications (such as multimedia processing, machine learning, and big data analytics) can often naturally tolerate soft errors.

We present Rely, a programming language that enables developers to reason about the quantitative reliability of an application – namely, the probability that it produces the correct result when executed on unreliable hardware. Rely allows developers to specify the reliability requirements for each value that a function produces.

We present a static quantitative reliability analysis that verifies quantitative requirements on the reliability of an application, enabling developers to perform sound and verify reliability engineering. The analysis takes a Rely program with a reliability specification and a hardware specification that characterizes the reliability of the underlying hardware components and verifies that the program satisfies its reliability specification when executed on the underlying unreliable hardware platform. We demonstrate the application of quantitative reliability analysis on six computations implemented in Rely.

Categories and Subject Descriptors F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs

Decision–Making with Complex Data Structures using Probabilistic Programming

Brian E. Ruttenberg and Avi Pfeffer
Charles River Analytics
625 Mt Auburn St
Cambridge MA 02140
{brutenberg, apfeffer}@cra.com

Abstract

Existing decision-theoretic reasoning frameworks such as decision networks use simple data structures and processes. However, decisions are often made based on complex data structures, such as social networks and protein sequences, and rich processes involving those structures. We present a framework for representing decision problems with complex data structures using probabilistic programming, allowing probabilistic models to be created with programming language constructs such as data structures and control flow. We provide a way to use arbitrary data types with minimal effort from the user, and an approximate decision-making algorithm that is effective even when the information space is very large or infinite. Experimental results show our algorithm working on problems with very large information spaces.

Introduction

ditional probability tables. In our problem, however, the protein and DNA sequence data structures are complex, as are the processes by which proteins map to DNA and the rate of DNA mutation. This presents two challenges: first, how do we represent decision problems with complex data structures, and second, how do we reason with them to create a policy that recommends the best decisions?

We address these challenges using probabilistic programming, which provides the ability to create probabilistic models using programming language constructs such as data structures and control flow. Probabilistic programming languages contain general purpose reasoning algorithms that can reason on all models written in the language. Probabilistic programming languages can naturally be extended with constructs denoting decisions, similar to the way decision networks extend Bayesian networks. By providing a general-purpose decision-making algorithm, all the benefits that probabilistic programming bestows upon standard probabilistic models are obtained for decision-theoretic models.
Content

Introduction

Quantitative Separation Logic

Case Studies

Conclusion
Modelling the Heap

Program states

- **Stacks** (variable valuations): \( \text{Stack} := \{ s \mid s : \text{Var} \rightarrow \mathbb{Z} \} \)
- **Heaps**: \( \text{Heap} := \{ h : \mathbb{N}_{>0} \rightarrow \mathbb{Z} \mid |\text{dom}(h)| < \infty \} \)
- **(Program) states**: \( \Sigma := \text{Stack} \times \text{Heap} \)

![Diagram showing stack and heap states](image)
**Operations**

- \( x := \text{new}(A) \)
- \( \text{free}(A) \)
- \( x := \langle A \rangle \)
- \( \langle A \rangle := A' \)

---

**Operational semantics**

\[
\begin{align*}
\text{alloc} & : \quad l \in \mathbb{N}_{>0} \setminus \text{dom}(h) \quad s(A) = z \\
\langle x := \text{new}(A), (s, h) \rangle & \rightarrow \langle \downarrow, (s[x \mapsto l], h[l \mapsto z]) \rangle \\
\text{free} & : \quad s(A) = l \in \text{dom}(h) \\
\langle \text{free}(A), (s, h) \rangle & \rightarrow \langle \downarrow, (s, h[l \mapsto \bot]) \rangle \\
\text{free-f} & : \quad s(A) \notin \text{dom}(h) \\
\langle \text{free}(A), (s, h) \rangle & \rightarrow \langle \mathcal{F}, (s, h) \rangle \\
\text{lookup} & : \quad s(A) = l \in \text{dom}(h) \\
\langle x := \langle A \rangle, (s, h) \rangle & \rightarrow \langle \downarrow, (s[x \mapsto h(l)], h) \rangle \\
\end{align*}
\]
Elementary Predicates

\[(s, h) \models \text{emp} \quad \text{iff} \quad \text{dom}(h) = \emptyset\]
\[(s, h) \models A \leftrightarrow A' \quad \text{iff} \quad \text{dom}(h) = \{s(A)\} \quad \text{and} \quad h(s(A)) = s(A')\]
\[(s, h) \models A \rightarrow \quad \text{iff} \quad \text{dom}(h) = \{s(A)\}\]
\[(s, h) \models A \leftrightarrow A' \quad \text{iff} \quad \text{dom}(h) \ni \{s(A)\} \quad \text{and} \quad h(s(A)) = s(A')\]
Classical Deficiency of Hoare Logic

The rule of constancy in Hoare Logic

\[
\begin{align*}
\{ P \} \ C \ \{ Q \} \quad & \quad \text{Var}(F) \cap \text{Mod}(C) = \emptyset \\
\{ P \land F \} \ C \ \{ Q \land F \}
\end{align*}
\]

becomes unsound when we consider pointers:

\[
\{ x \mapsto \rightarrow 0 \} \langle x \rangle := 1
\]

\[
\{ x \mapsto \rightarrow 0 \land y \mapsto \rightarrow 0 \} \langle x \rangle := 1
\]

is not valid (because \( y \) could alias \( x \)).
Classical Deficiency of Hoare Logic

The rule of constancy in Hoare Logic

\[ \{ P \} \ C \ \{ Q \} \quad \text{Var}(F) \cap \text{Mod}(C) = \emptyset \]

\[ \{ P \land F \} \ C \ \{ Q \land F \} \]

becomes unsound when we consider pointers:

\[ \{ x \mapsto 0 \} \langle x \rangle := 1 \{ x \mapsto 1 \} \]

\[ \{ x \mapsto 0 \land y \mapsto 0 \} \langle x \rangle := 1 \{ x \mapsto 1 \land y \mapsto 0 \} \]

is not valid (because \( y \) could alias \( x \))
Solution: Separating Conjunction

Separating conjunction

New conjunction operator ★ meaning “and, separately in memory”

For example, \( x \stackrel{\leftrightarrow}{\implies} y \star y \stackrel{\leftrightarrow}{\implies} x \):
Solution: Separating Conjunction

Separating conjunction

New conjunction operator $\star$ meaning “and, separately in memory”

For example, $x \mapsto y \star y \mapsto x$:

Semantics:

$(s, h) \models F_1 \star F_2$ iff $\exists h_1, h_2 \in Heap : h = h_1 \cup h_2, (s, h_1) \models F_1$ and $(s, h_2) \models F_2$
A Proper Rule for Local Reasoning

Frame rule of Separation Logic:

\[
\begin{array}{c}
\{ P \} C \{ Q \} \quad \text{Var}(F) \cap \text{Mod}(C) = \emptyset \\
\{ P \star F \} C \{ Q \star F \}
\end{array}
\]
A Proper Rule for Local Reasoning

Frame rule of Separation Logic:

\[
\begin{align*}
\{ P \} \ C \ \{ Q \} \quad \text{Var}(F) \cap \text{Mod}(C) &= \emptyset \\
\{ P \star F \} \ C \ \{ Q \star F \}
\end{align*}
\]

In particular,

\[
\begin{align*}
\{ x \mapsto 0 \} \langle x \rangle &:= 1 \{ x \mapsto 1 \} \\
\{ x \mapsto 0 \star y \mapsto 0 \} \langle x \rangle &:= 1 \{ x \mapsto 1 \star y \mapsto 0 \}
\end{align*}
\]

is valid as \( \star \) excludes aliasing of \( x \) and \( y \).
Example (Recursive tree disposal)

Binary trees: \( \text{tree}(x) := (x = 0 \land \text{emp}) \lor \exists y, z : x \mapsto (y, z) \ast \text{tree}(y) \ast \text{tree}(z) \)
Example (Recursive tree disposal)

Binary trees: $\text{tree}(x) := (x = 0 \land \text{emp}) \lor \exists y, z : x \mapsto (y, z) \ast \text{tree}(y) \ast \text{tree}(z)$

$$\text{delete}(x) \{$$
  $$\text{if } x = 0 \text{ then return}$$
  $$\text{else } \{$$
    $$l := \langle x \rangle; r := \langle x + 1 \rangle;$$
    $$\text{delete}(l);$$
    $$\text{delete}(r);$$
    $$\text{free}(x); \text{free}(x + 1)$$
  $$\}$$
$$\}$$
The Power of Separating Conjunction

Example (Recursive tree disposal)

Binary trees: \( \text{tree}(x) := (x = 0 \land \text{emp}) \lor \exists y, z : x \mapsto (y, z) \star \text{tree}(y) \star \text{tree}(z) \)

\[
\begin{align*}
delete(x) \{ & \{ \text{tree}(x) \} \\
& \begin{cases}
if x = 0 & \text{then return } \{ \text{emp} \} \\
else & \{ \exists y, z : x \mapsto (y, z) \star \text{tree}(y) \star \text{tree}(z) \}
\end{cases} \\
& \quad l := \langle x \rangle; r := \langle x + 1 \rangle; \\
& \quad \{ x \mapsto (l, r) \star \text{tree}(l) \star \text{tree}(r) \} \\
& \quad \text{delete}(l); \\
& \quad \{ x \mapsto (l, r) \star \text{emp} \star \text{tree}(r) \} \\
& \quad \text{delete}(r); \\
& \quad \{ x \mapsto (l, r) \star \text{emp} \star \text{emp} \} \\
& \quad \text{free}(x); \text{free}(x + 1) \\
& \quad \{ \text{emp} \star \text{emp} \star \text{emp} \} \\
& \quad \{ \text{emp} \} \\
& \quad \{ \text{emp} \}
\end{align*}
\]
Probabilistic Weakest Preconditions

[Deuter Kozen, Annabelle McIver, Carroll Morgan]

Content

Introduction

Separation Logic [John Reynolds, Peter O’Hearn]

Probabilistic Weakest Preconditions [Deuter Kozen, Annabelle McIver, Carroll Morgan]

Quantitative Separation Logic

Case Studies

Conclusion
Use an inductively defined backwards moving predicate transformer

\[ \text{wp} [\mathcal{C}] : 2^\Sigma \rightarrow 2^\Sigma \]
Weakest Precondition Reasoning

[Edsger W. Dijkstra]

Use an inductively defined backwards moving predicate transformer

\[ \text{wp} \left[ C \right] : 2^\Sigma \rightarrow 2^\Sigma \]

postcondition \( Q \) evaluated in final states after termination of \( C_2 \)

weakest precondition of \( Q \) w.r.t. \( C_2 \)

weakest precondition of \( \text{wp} \left[ C \right] \) w.r.t. \( C_1 \) or in other words:

weakest precondition of \( Q \) w.r.t. \( C_1 \); \( C_2 \)

evaluated in initial states before executing \( C_1 \); \( C_2 \)
Weakest Precondition Reasoning

Use an inductively defined backwards moving predicate transformer

\[ \text{wp} \left[ C \right] : 2^\Sigma \rightarrow 2^\Sigma \]
Weakest Precondition Reasoning

Use an inductively defined backwards moving predicate transformer

$$\text{wp } [C] : 2^\Sigma \rightarrow 2^\Sigma$$

C1; wp[C2](Q) C2; Q

weakest precondition of Q w.r.t. C2
evaluated in initial states before executing C2

postcondition Q evaluated in final states after termination of C2
Weakest Precondition Reasoning

[Edsger W. Dijkstra]

Use an inductively defined backwards moving predicate transformer

\[ \text{wp} \left[ C \right] : 2^\Sigma \rightarrow 2^\Sigma \]

\[ \text{wp} \left[ C_1 \right] (\text{wp} \left[ C_2 \right] (Q)) \]

\[ C_1 ; \]

\[ \text{wp} \left[ C_2 \right] (Q) \]

\[ C_2 \]

\[ Q \]

weakest precondition of \( Q \)

w.r.t. \( C_2 \)

evaluated in initial states before executing \( C_2 \)

postcondition \( Q \)

evaluated in final states after termination of \( C_2 \)
Weakest Precondition Reasoning

Use an inductively defined backwards moving predicate transformer

\[ \text{wp} \left[ C \right] : 2^\Sigma \rightarrow 2^\Sigma \]
Weakest Precondition Reasoning

Use an inductively defined backwards moving predicate transformer

\[ \text{wp} \left[ C \right] : 2^\Sigma \rightarrow 2^\Sigma \]

\[ \text{wp} \left[ C_1 \right] \left( \text{wp} \left[ C_2 \right] \left( Q \right) \right) \]

weakest precondition of \( \text{wp} \left[ C_2 \right] \left( Q \right) \)
w.r.t. \( C_1 \)
or in other words:
weakest precondition of \( Q \)
w.r.t. \( C_1 ; C_2 \)

\[ C_1 ; \]

\[ \text{wp} \left[ C_2 \right] \left( Q \right) \]

weakest precondition of \( Q \)
w.r.t. \( C_2 \)
evaluated in initial states before executing \( C_2 \)

\[ C_2 \]

\[ Q \]

postcondition \( Q \)
evaluated in final states after termination of \( C_2 \)
Weakest Precondition Reasoning

Use an inductively defined backwards moving predicate transformer

\[ \text{wp} [C] : 2^\Sigma \rightarrow 2^\Sigma \]

- weakest precondition of \( Q \) w.r.t. \( C_1 \) evaluated in initial states before executing \( C_1 ; C_2 \)
- weakest precondition of \( Q \) w.r.t. \( C_2 \) evaluated in initial states before executing \( C_2 \)
- postcondition \( Q \) evaluated in final states after termination of \( C_2 \)
Example

1. Let program $C_1$ be

   $\{ x := 5 \}^{4/5} \{ x := 10 \}$

   The expected value of $x$ on $C_1$’s termination is

   $\frac{4}{5} \cdot 5 + \frac{1}{5} \cdot 10 = 6$

2. Let program $C_2$ be

   $\{ x := x + 5 \}^{4/5} \{ x := 10 \}$

   The expected value of $x$ on $C_2$’s termination is (for $x_0 =$ initial value of $x$)

   $\frac{4}{5} \cdot (x_0 + 5) + \frac{1}{5} \cdot 10$

3. The probability that $x = 10$ on $C_2$’s termination is

   $\frac{4}{5} \cdot \{ x_0 + 5 = 10 \} + \frac{1}{5} \cdot 1$
Example

1. Let program $C_1$ be

$$\{ x := 5 \}[\frac{4}{5}] \{ x := 10 \}$$

The expected value of $x$ on $C_1$’s termination is

$$\frac{4}{5} \cdot 5 + \frac{1}{5} \cdot 10 = 6$$

2. Let program $C_2$ be

$$\{ x := x + 5 \}[\frac{4}{5}] \{ x := 10 \}$$

The expected value of $x$ on $C_2$’s termination is (for $x_0 = \text{initial value of } x$)

$$\frac{4}{5} \cdot (x_0 + 5) + \frac{1}{5} \cdot 10 = \frac{4}{5} \cdot x_0 + 6$$
Example

1. Let program $C_1$ be

$$\{ x := 5 \} \left[ \frac{4}{5} \right] \{ x := 10 \}$$

The expected value of $x$ on $C_1$’s termination is

$$\frac{4}{5} \cdot 5 + \frac{1}{5} \cdot 10 = 6$$

2. Let program $C_2$ be

$$\{ x := x + 5 \} \left[ \frac{4}{5} \right] \{ x := 10 \}$$

The expected value of $x$ on $C_2$’s termination is (for $x_0 = \text{initial value of } x$)

$$\frac{4}{5} \cdot (x_0 + 5) + \frac{1}{5} \cdot 10 = \frac{4}{5} \cdot x_0 + 6$$

3. The probability that $x = 10$ on $C_2$’s termination is

$$\frac{4}{5} \cdot [x_0 + 5 = 10] + \frac{1}{5} \cdot 1 = \frac{4}{5} \cdot [x_0 = 5] + \frac{1}{5}$$
Expectations

Classical predicates

- $F : \Sigma \rightarrow \mathbb{B}$
- $F \leq G \iff F \Rightarrow G$
Expectations

Classical predicates

- \( F : \Sigma \rightarrow \mathbb{B} \)
- \( F \leq G \iff F \implies G \)

Quantitative predicates: expectations (\(\neq\) expectations in probability theory – read as: random variable)

- \( E = \{ f \mid f : \Sigma \rightarrow \mathbb{R}_{\geq 0} \} \)
- \( f \leq g \iff \forall \sigma \in \Sigma : f(\sigma) \leq g(\sigma) \)
- complete lattice
The Weakest Preexpectation Transformer

- Given: Probabilistic program $C$ and (post)expectation $f$
The Weakest Preexpectation Transformer  

- **Given**: Probabilistic program $C$ and (post)expectation $f$
- **Question**: What is the expected value of $f$ after successful termination of $C$?
The Weakest Preexpectation Transformer  

- **Given:** Probabilistic program $C$ and (post)expectation $f$
- **Question:** What is the expected value of $f$ after successful termination of $C$?
The Weakest Preexpectation Transformer

[Cozen 1983, McIver 1999, McIver & Morgan 2005]

- Given: Probabilistic program $C$ and (post)expectation $f$
- Question: What is the expected value of $f$ after successful termination of $C$?

\[
\text{Weakest preexpectation: Mapping from initial state } \sigma \text{ to expected value of } f \text{ evaluated in final states reached after successful termination of } C \]

\[
\wp_{J_{C \Sigma}}(f) = \lambda \sigma \int_{\Sigma} f \, dJ_{C \Sigma} \quad \sigma \in \mathcal{E} = \{ f \mid f : \Sigma \rightarrow \mathbb{R} \}_{\geq 0}
\]
The Weakest Preexpectation Transformer  

- **Given**: Probabilistic program $C$ and (post)expectation $f$
- **Question**: What is the expected value of $f$ after successful termination of $C$?

$$\text{Exp} \left[ f(\sigma_1) \quad f(\sigma_2) \quad f(\sigma_3) \quad \cdots \right]$$
The Weakest Preexpectation Transformer


- **Given:** Probabilistic program $C$ and (post)expectation $f$
- **Question:** What is the expected value of $f$ after successful termination of $C$?

\[ \text{Weakest preexpectation: Mapping from initial state } \sigma \text{ to expected value of } f \text{ evaluated in final states reached after successful termination of } C \]

\[ \text{wp} \] \[ J \]

\[ K \]

\[ (f) = \lambda \sigma . \int_{\Sigma} f \, dJ_{C \sigma} \]

\[ \sigma' \]

\[ \Sigma = \{ f \mid f : \Sigma \rightarrow \mathbb{R}^{\infty} \geq 0 \} \]

(with $J_{C \sigma}$ the distribution over final states when running $C$ on state $\sigma$)
The Weakest Preexpectation Transformer

• Given: Probabilistic program $C$ and (post)expectation $f$
• Question: What is the expected value of $f$ after successful termination of $C$?
The Weakest Preexpectation Transformer


- **Given:** Probabilistic program $C$ and (post)expectation $f$
- **Question:** What is the expected value of $f$ after successful termination of $C$?

\[
\begin{align*}
\text{Exp} \left[ f(\sigma_1) \right] & \quad \text{Exp} \left[ f(\sigma_2) \right] & \quad \text{Exp} \left[ f(\sigma_3) \right] & \quad \cdots \\
& \quad & \quad & \\
\sigma & \quad C & \quad & \\
\sigma' & \quad C & \quad & \\
\end{align*}
\]
The Weakest Preexpectation Transformer

- **Given:** Probabilistic program $C$ and (post)expectation $f$
- **Question:** What is the expected value of $f$ after successful termination of $C$?

**Weakest preexpectation:** Mapping from initial state $\sigma$ to expected value of $f$ evaluated in final states reached after successful termination of $C$ on $\sigma$

$$\text{wp } [C](f) = \lambda_\sigma. \int_{\Sigma} f \ d[\Sigma]_\sigma \in E = \{ f \mid f : \Sigma \to \mathbb{R}_{\geq 0}^\infty \}$$

(with $[C]_\sigma$ the distribution over final states when running $C$ on state $\sigma$)
### The Weakest Preexpectation Calculus

<table>
<thead>
<tr>
<th>$C$</th>
<th>$\text{wp} \left[ [C] (f) \right]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip</td>
<td>$f$</td>
</tr>
<tr>
<td>$x := A$</td>
<td>$f [x/A]$</td>
</tr>
<tr>
<td>$C_1 ; C_2$</td>
<td>$\text{wp} \left[ [C_1] \left( \text{wp} \left[ [C_2] (f) \right) \right) \right]$</td>
</tr>
<tr>
<td>if $(B) { C_1 } \text{ else } { C_2 }$</td>
<td>$[B] \cdot \text{wp} \left[ [C_1] (f) \right] + [\neg B] \cdot \text{wp} \left[ [C_2] (f) \right]$</td>
</tr>
<tr>
<td>${ C_1 } [p] { C_2 }$</td>
<td>$p \cdot \text{wp} \left[ [C_1] (f) \right] + (1 - p) \cdot \text{wp} \left[ [C_2] (f) \right]$</td>
</tr>
<tr>
<td>while $(B) { C' }$</td>
<td>$\text{lfp } X. \left[ [\neg B] \cdot f + [B] \cdot \text{wp} \left[ [C'] (X) \right] \right]$</td>
</tr>
</tbody>
</table>

**Loop unrolling**

---

**loop unrolling**
Weakest Preexpectations

Example

1. Let program $C_1$ be $\{ x := 5 \} [4/5] \{ x := 10 \}$. For $f_1 = x (:= \lambda \sigma. \sigma(x))$, we have

$$wp [C_1] (f_1) = \frac{4}{5} \cdot wp [x := 5] (f_1) + \frac{1}{5} \cdot wp [x := 10] (f_1) = \frac{4}{5} \cdot 5 + \frac{1}{5} \cdot 10 = 6$$
Weakest Preexpectations

Example

1. Let program $C_1$ be $\{ x := 5 \} \left[ \frac{4}{5} \right] \{ x := 10 \}$. For $f_1 = x := \lambda \sigma. \sigma(x)$, we have

$$\text{wp}\left[ C_1 \right] (f_1) = \frac{4}{5} \cdot \text{wp}\left[ x := 5 \right] (f_1) + \frac{1}{5} \cdot \text{wp}\left[ x := 10 \right] (f_1) = \frac{4}{5} \cdot 5 + \frac{1}{5} \cdot 10 = 6$$

2. Let program $C_2$ be $\{ x := x + 5 \} \left[ \frac{4}{5} \right] \{ x := 10 \}$. Here

$$\text{wp}\left[ C_2 \right] (f_1) = \frac{4}{5} \cdot \text{wp}\left[ x := x + 5 \right] (f_1) + \frac{1}{5} \cdot \text{wp}\left[ x := 10 \right] (f_1)$$

$$= \frac{4}{5} \cdot (x + 5) + \frac{1}{5} \cdot 10 = \frac{4}{5} \cdot x + 6$$
Weakest Preexpectations

Example

1. Let program $C_1$ be $\{ x := 5 \} \left[ \frac{4}{5} \right] \{ x := 10 \}$. For $f_1 = x := \lambda \sigma. \sigma(x)$, we have

$$wp \left[ C_1 \right] (f_1) = \frac{4}{5} \cdot wp \left[ x := 5 \right] (f_1) + \frac{1}{5} \cdot wp \left[ x := 10 \right] (f_1) = \frac{4}{5} \cdot 5 + \frac{1}{5} \cdot 10 = 6$$

2. Let program $C_2$ be $\{ x := x + 5 \} \left[ \frac{4}{5} \right] \{ x := 10 \}$. Here

$$wp \left[ C_2 \right] (f_1) = \frac{4}{5} \cdot wp \left[ x := x + 5 \right] (f_1) + \frac{1}{5} \cdot wp \left[ x := 10 \right] (f_1)$$

$$= \frac{4}{5} \cdot (x + 5) + \frac{1}{5} \cdot 10 = \frac{4}{5} \cdot x + 6$$

3. For $f_2 = [x = 10]$, we have

$$wp \left[ C_2 \right] (f_2) = \frac{4}{5} \cdot wp \left[ x := x + 5 \right] (f_2) + \frac{1}{5} \cdot wp \left[ x := 10 \right] (f_2)$$

$$= \frac{4}{5} \cdot [x + 5 = 10] + \frac{1}{5} \cdot [10 = 10] = \frac{4}{5} \cdot [x = 5] + \frac{1}{5}$$
Quantitative Separation Logic

Content

Introduction

Separation Logic [John Reynolds, Peter O’Hearn]

Probabilistic Weakest Preconditions [Dexter Kozen, Annabelle McIver, Carroll Morgan]

Quantitative Separation Logic

Case Studies

Conclusion
States and expectations

- **States**: $\Sigma := \text{Stack} \times \text{Heap} = \{(s, h) \mid s : \text{Var} \to \mathbb{Z}, h : \mathbb{N}_{>0} \to \mathbb{Z}, |\text{dom}(h)| < \infty\}$
- **Expectations**: $\mathcal{E} := \{f \mid f : \Sigma \to \mathbb{R}_{\geq 0}\}$

Example

- Postexpectation $f$
- Weakest preexpectation $\wp$
- $J_C^K(f)$

1. Probability of memory-safe termination
2. Expected square value of variable $x$
3. Probability of terminating with empty heap
4. Expected size of heap

26 of 38  Quantitative Separation Logic
Thomas Noll
November 7, 2019
States and expectations

- **States:** $\Sigma := Stack \times Heap = \{(s, h) \mid s : \text{Var} \rightarrow \mathbb{Z}, h : \mathbb{N}_{>0} \rightarrow \mathbb{Z}, |\text{dom}(h)| < \infty\}$

- **Expectations:** $\mathbb{E} := \{f \mid f : \Sigma \rightarrow \mathbb{R}_{\geq 0}\}$

Example

<table>
<thead>
<tr>
<th>Postexpectation $f$</th>
<th>Weakest preexpectation $wp \llbracket C \rrbracket (f)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1 := \lambda(s, h).1$</td>
<td>Probability of memory-safe termination</td>
</tr>
<tr>
<td>$x^2 := \lambda(s, h). s(x)^2$</td>
<td>Expected square value of variable $x$</td>
</tr>
<tr>
<td>$\text{emp}$</td>
<td>Probability of terminating with empty heap</td>
</tr>
<tr>
<td>$\text{size} := \lambda(s, h).</td>
<td>\text{dom}(h)</td>
</tr>
</tbody>
</table>
Quantitative Conjunction

Classical conjunction

\[ F \land G \]
### Quantitative Conjunction

#### Classical conjunction

\[ F \land G \]

#### Quantitative conjunction

\[ f \cdot g = \lambda(s, h). \ f(s, h) \cdot g(s, h) \]
Quantitative Conjunction

Classical conjunction

\[ F \land G \]

Quantitative conjunction

\[ f \cdot g = \lambda(s, h). \ f(s, h) \cdot g(s, h) \]

Corollary

\[ [F \land G] = [F] \cdot [G] \]
Classical separating conjunction

\[(s, h) \models F * G \iff \exists h_1, h_2 : h = h_1 \uplus h_2, (s, h_1) \models F \text{ and } (s, h_2) \models G\]
Classical separating conjunction

\[(s, h) \models F \star G \iff \exists h_1, h_2 : h = h_1 \sqcup h_2, (s, h_1) \models F \text{ and } (s, h_2) \models G\]
Separating Conjunction

Classical separating conjunction

\[(s, h) \models F \star G \iff \exists h_1, h_2 : h = h_1 \uplus h_2, (s, h_1) \models F \text{ and } (s, h_2) \models G\]

Quantitative separating conjunction

\[f \star g = \lambda(s, h). \max \{f(s, h_1) \cdot g(s, h_2) \mid h = h_1 \uplus h_2\}\]
Separating Implication

Classical separating implication

\[(s, h) \models F \rightarrowstar G \iff \forall h' \text{ with } h' \perp h \text{ and } (s, h') \models F : (s, h \uplus h') \models G\]
Classical separating implication

\[(s, h) \models F \rightarrow^\star G \iff \forall h' \text{ with } h' \perp h \text{ and } (s, h') \models F : (s, h \sqcup h') \models G\]

Quantitative separating implication

\[f \rightarrow^\star g = \lambda(s, h). \inf \left\{ \frac{g(s, h \sqcup h')}{f(s, h')} \mid \begin{array}{l} h' \perp h \text{ and } f(s, h') > 0 \text{ and} \\ \text{not } f(s, h') = \infty = g(s, h \star h') \end{array} \right\} \]
## Separating Implication

### Classical separating implication

\[(s, h) \models F \to G \iff \forall h' \text{ with } h' \perp h \text{ and } (s, h') \models F : (s, h \mathbin{\cup} h') \models G\]

### Quantitative separating implication

\[f \to g = \lambda(s, h). \inf \left\{ \frac{g(s, h \mathbin{\cup} h')}{f(s, h')} \middle| \begin{array}{l} h' \perp h \text{ and } f(s, h') > 0 \text{ and} \hfill \\
\text{not } f(s, h') = \infty = g(s, h \star h') \end{array} \right\}\]

### Corollary

\[[F] \to g = \lambda(s, h). \inf \{ g(s, h \mathbin{\cup} h') \middle| h' \perp h \text{ and } (s, h') \models F \} \]
Weakest Preexpectation for Memory Allocation

\[
\begin{array}{c|ccc}
\text{s:} & x & y & \ldots \\
\hline
u & w & & \\
\end{array}
\quad
\begin{array}{c|ccc}
\text{h:} & \alpha & \beta & \gamma & \ldots \\
\hline
a & b & c & & \\
\end{array}
\]
Weakest Preexpectation for Memory Allocation

\[
x := \text{new} (A)
\]

\[
s : \\
x : y : \ldots \\
u \quad w
\]

\[
h : \\
\alpha : \beta : \gamma : \ldots \\
 a \quad b \quad c
\]
Weakest Preexpectation for Memory Allocation

\[ x := \text{new}(A) \]

\[
\begin{array}{c|c|c|c|c|c}
  & x & y & \cdots & & \\
\hline
  s: & \alpha & \beta & \gamma & \cdots & \\
  u & a & b & c & \cdots & \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c}
  & x & y & \cdots & & \\
\hline
  s_{\perp}: & \alpha & \beta & \gamma & \cdots & \\
  v & a & b & c & \cdots & \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c}
  & x & y & \cdots & & \\
\hline
  h_{\perp}: & \alpha & \beta & \gamma & \cdots & \\
  w & a & b & c & \cdots & \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c}
  & x & y & \cdots & & \\
\hline
  v: & a & b & c & \cdots & \\
  s_{\perp}(A) & & & & & \\
\end{array}
\]
Weakest Preexpectation for Memory Allocation

\[ x := \text{new}(A) \]

\[
\langle x := \text{new}(A), (s, h) \rangle \quad \langle v = 1, (s[x/1], h \ast \{1 \mapsto s(A)\}) \rangle
\]
Weakest Preexpectation for Memory Allocation

\[ x := \text{new}(A) \]

\[ s: \]
\[
\begin{array}{ccc}
  x & y & \ldots \\
  u & w & \alpha & \beta & \gamma & \ldots \\
\end{array}
\]

\[ h: \]
\[
\begin{array}{ccc}
  a & b & c & \ldots \\
\end{array}
\]

\[ s_\downarrow: \]
\[
\begin{array}{ccc}
  x & y & \ldots \\
  v & w & a & b & c & \ldots \\
\end{array}
\]

\[ h_\downarrow: \]
\[
\begin{array}{ccc}
  \alpha & \beta & \gamma & \ldots \\
  v & w & a & b & c & \ldots \\
\end{array}
\]

\[ \nu = 1 \]
\[ \downarrow, \langle s[x/1], h \ast \{1 \mapsto s(A)\} \rangle \]

\[ \nu = 2 \]
\[ \downarrow, \langle s[x/2], h \ast \{2 \mapsto s(A)\} \rangle \]
Weakest Preexpectation for Memory Allocation

\[
x := \text{new}(A)
\]

\[
\begin{array}{c|c|c|c|}
  s: & x & y & \ldots \\
  \alpha & \beta & \gamma & \ldots \\
  u & w
\end{array}
\quad
\begin{array}{c|c|c|c|}
  h: & \alpha & \beta & \gamma & \ldots \\
  a & b & c & \ldots \\
  v & w
\end{array}
\]

\[
\begin{array}{c|c|c|c|}
  s_{\downarrow}: & x & y & \ldots \\
  \alpha & \beta & \gamma & \ldots \\
  v & w
\end{array}
\quad
\begin{array}{c|c|c|c|}
  h_{\downarrow}: & \alpha & \beta & \gamma & \ldots \\
  a & b & c & \ldots \\
  s(A)
\end{array}
\]

\[
\begin{array}{c|c|c|c|}
  v = 1 & \langle \downarrow, (s[x/1], h \ast \{1 \mapsto s(A)\}) \rangle
\end{array}
\]

\[
\begin{array}{c|c|c|c|}
  v = 2 & \langle \downarrow, (s[x/2], h \ast \{2 \mapsto s(A)\}) \rangle
\end{array}
\]

\[
\vdots
\]

infinite branching!
Weakest Preexpectation for Memory Allocation

\[ x := \text{new}(A) \]

\[ s : x \quad y \quad \ldots \quad u \quad w \]

\[ h : \alpha \quad \beta \quad \gamma \quad \ldots \quad a \quad b \quad c \]

\[ s_{\downarrow} : v \quad w \quad \ldots \quad a \quad b \quad c \quad \ldots \]

\[ h_{\downarrow} : \beta \quad \gamma \quad \ldots \quad \ast \quad v : s(A) \]

\[ \text{wp}[x := \text{new}(A)] \]

\[ f \]

\[ \langle \downarrow, (s[x/1], h \star \{1 \mapsto s(A)\}) \rangle \]

\[ \langle x := \text{new}(A), (s, h) \rangle \]

\[ \langle \downarrow, (s[x/2], h \star \{2 \mapsto s(A)\}) \rangle \]

\[ \vdots \]

\[ \text{infinite branching!} \]
Weakest Preexpectation for Memory Allocation

\[ x := \text{new}(A) \]

\[ [v \mapsto A] \rightarrow f[x/v] \]

\[ \text{wp}[x := \text{new}(A)] \]

\[ \langle \downarrow, (s[x/1], h \cdot \{1 \mapsto s(A)\}) \rangle \]

\[ v = 1 \]

\[ \langle x := \text{new}(A), (s, h) \rangle \]

\[ v = 2 \]

\[ \langle \downarrow, (s[x/2], h \cdot \{2 \mapsto s(A)\}) \rangle \]

\[ \vdots \]

infinite branching!
Weakest Preexpectation for Memory Allocation

\[ x := \text{new} (A) \]

\[
\begin{array}{c|c|c}
\hline
x & y & \ldots \\
\hline
u & w & \alpha & \beta & \gamma & \ldots \\
\hline
\end{array}
\]

\[ \inf_{v \in \mathbb{N}} [v \mapsto A] \rightarrow f[x/v] \]

\[ \text{wp} [x := \text{new} (A)] \]

\[ \langle \downarrow, (s[x/1], h \star \{1 \mapsto s(A)\}) \rangle \]

\[ \langle x := \text{new} (A), (s, h) \rangle \]

\[ \langle \downarrow, (s[x/2], h \star \{2 \mapsto s(A)\}) \rangle \]

\[ \vdots \]

infinite branching!
### WP Rules for Pointer Operations

#### Classical WP rules for formula $F$ in SL

<table>
<thead>
<tr>
<th>$C$</th>
<th>$\text{wp} [C] (F)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x := \text{new} (A)$</td>
<td>$\forall v. v \mapsto A \rightarrow F [x/v] $</td>
</tr>
<tr>
<td>$\text{free}(A)$</td>
<td>$A \mapsto \neg \star F$</td>
</tr>
<tr>
<td>$x := \langle A \rangle$</td>
<td>$\exists v. A \mapsto v \star (A \mapsto v \rightarrow F [x/v])$</td>
</tr>
<tr>
<td>$\langle A \rangle := A'$</td>
<td>$A \mapsto \neg \star (A \mapsto A' \rightarrow F)$</td>
</tr>
</tbody>
</table>
WP Rules for Pointer Operations

### Classical WP rules for formula $F$ in SL

<table>
<thead>
<tr>
<th>$C$</th>
<th>$\text{wp} \left[ C \right] (F)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x := \text{new}(A)$</td>
<td>$\forall v. v \mapsto A \rightarrow F[x/v]$</td>
</tr>
<tr>
<td>$\text{free}(A)$</td>
<td>$A \mapsto \neg \star F$</td>
</tr>
<tr>
<td>$x := \langle A \rangle$</td>
<td>$\exists v. A \mapsto v \star (A \mapsto v \rightarrow F[x/v])$</td>
</tr>
<tr>
<td>$\langle A \rangle := A'$</td>
<td>$A \mapsto \neg \star (A \mapsto A' \rightarrow F)$</td>
</tr>
</tbody>
</table>

### Quantitative WP rules for expectation $f$ in QSL

<table>
<thead>
<tr>
<th>$C$</th>
<th>$\text{wp} \left[ C \right] (f)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x := \text{new}(A)$</td>
<td>$\inf_{v \in \mathbb{N}} [v \mapsto A] \rightarrow f[x/v]$</td>
</tr>
<tr>
<td>$\text{free}(A)$</td>
<td>$[A \mapsto \neg] \star f$</td>
</tr>
<tr>
<td>$x := \langle A \rangle$</td>
<td>$\sup_{v \in \mathbb{Z}} [A \mapsto v] \star ([A \mapsto v] \rightarrow f[x/v])$</td>
</tr>
<tr>
<td>$\langle A \rangle := A'$</td>
<td>$[A \mapsto \neg] \star ([A \mapsto A'] \rightarrow f)$</td>
</tr>
</tbody>
</table>
A Small Example

Example (Probabilistic list extension)

\[
c := 1;
\]

\[
\text{while}(c = 1) \{
\{ c := 0 \} \left[ \frac{1}{2} \right] \{ x := \text{new}(x) \}
\}
\]
A Small Example

Example (Probabilistic list extension)

\[
c := 1;
\]

\[
\text{while}(c = 1)\{
\{
\text{c := 0}\} \left[\frac{1}{2}\right] \{ x := \text{new}(x) \}
\}
// size
\]
A Small Example

Example (Probabilistic list extension)

```
c := 1;

while (c = 1) {
  { c := 0 } [1/2] { x := new(x) }
  // size + [c = 1]
}
// size
```
A Small Example

Example (Probabilistic list extension)

```c

\[
c := 1;
\]

while (c = 1) {
    // \( \frac{1}{2} \text{size} + \frac{1}{2} \inf_v [v \mapsto x] \rightarrow (\text{size} + [c = 1]) \)
    \[
    \{ c := 0 \} \left[ \frac{1}{2} \right] \{ x := \text{new} (x) \}
    // \text{size} + [c = 1]
    \]
}
// size
```
A Small Example

Example (Probabilistic list extension)

\[ c := 1; \]

\[ \text{while} (c = 1) \{ \]
\[ \quad \text{// } \text{size} + \frac{1}{2} \left( 1 + [c = 1] \right) \]
\[ \quad \text{// } \frac{1}{2} \text{size} + \frac{1}{2} \inf_{\nu} [\nu \mapsto x] \rightarrow (\text{size} + [c = 1]) \]
\[ \quad \{ c := 0 \} \left[ \frac{1}{2} \right] \{ x := \text{new} (x) \} \]
\[ \quad \text{// } \text{size} + [c = 1] \]
\[ \} \]

\[ \text{// } \text{size} \]
A Small Example

Example (Probabilistic list extension)

\[ c := 1; \]

\[
// [c \neq 1] \cdot \text{size} + [c = 1] \cdot (\text{size} + \frac{1}{2} (1 + [c = 1]))
\]

\[ \text{while}(c = 1)\{
\]

\[
// \text{size} + \frac{1}{2} (1 + [c = 1])
\]

\[
// \frac{1}{2} \text{size} + \frac{1}{2} \inf_v [v \mapsto x] \rightarrow (\text{size} + [c = 1])
\]

\[
\{ c := 0 \} [1/2] \{ x := \text{new}(x) \}
\]

\[
// \text{size}
\]

\[ \text{size} + [c = 1] \]

\[ \text{size} \]
A Small Example

Example (Probabilistic list extension)

```
c := 1;
// size + [c = 1]
// [c ≠ 1] · size + [c = 1] · (size + \frac{1}{2}(1 + [c = 1]))
while(c = 1){
    // size + \frac{1}{2}(1 + [c = 1])
    // \frac{1}{2}size + \frac{1}{2}\inf_{v} [v ↦ x] → (size + [c = 1])
    { c := 0 } [1/2] { x := new(x) }
    // size + [c = 1]
}
// size
```
A Small Example

Example (Probabilistic list extension)

```plaintext
// size + 1

c := 1;

// size + [c = 1]

// [c ≠ 1] · size + [c = 1] · (size + \frac{1}{2} (1 + [c = 1]))

while (c = 1) {
    // size + \frac{1}{2} (1 + [c = 1])
    // \frac{1}{2} size + \frac{1}{2} \inf_{v} [v \mapsto x] \mapsto (size + [c = 1])

    { c := 0 } [^1/2] { x := new(x) }
    // size + [c = 1]

    }

// size
```
Case Studies

Content

Introduction

Separation Logic  [John Reynolds, Peter O’Hearn]

Probabilistic Weakest Preconditions  [Dexter Kozen, Annabelle McIver, Carroll Morgan]

Quantitative Separation Logic

Case Studies

Conclusion
Example (Array randomisation)

```plaintext
// 1/n! · max{ [array ↦ (α_{π(0)}, ..., α_{π(n-1)})] | π permutation }

randomize(array, n) {
    i := 0;
    while (0 ≤ i < n) {
        j := uniform(i, n - 1);
        swap(array, i, j);
        i := i + 1
    }  

// [array ↦ (α_0, ..., α_{n-1})]
```
Example (Faulty garbage collector)

```
// [tree(x)] · (1 − p)₁/₂ · size
delete(x) {
  if (x ≠ 0) { // fails with probability p
    { skip } [p] {
      l := ⟨x⟩; r := ⟨x + 1⟩;
      delete(l);
      delete(r);
      free(x); free(x + 1)
    }
  }
}
// [emp]
```
Conclusion

Content

Introduction

Separation Logic [John Reynolds, Peter O’Hearn]

Probabilistic Weakest Preconditions [Dexter Kozen, Annabelle McIver, Carroll Morgan]

Quantitative Separation Logic

Case Studies

Conclusion
## Conclusion

### Quantitative Separation Logic

- as an assertion language
- as a verification system
- as a conservative, sound extension of separation logic

Further topics
- related soundness proofs
- handling of recursive procedures
- more case studies (lossy list reversal, ...)

Ongoing/future work
- certification in Isabelle/HOL
- support for continuous distributions
- support for concurrency
Conclusion

Quantitative Separation Logic

- as an assertion language
- as a verification system
- as a conservative, sound extension of separation logic

Further topics

- related soundness proofs
- handling of recursive procedures
- more case studies (lossy list reversal, ...)

[POPL 2019]
## Conclusion

### Quantitative Separation Logic

- as an assertion language
- as a verification system
- as a conservative, sound extension of separation logic

### Further topics [POPL 2019]

- related soundness proofs
- handling of recursive procedures
- more case studies (lossy list reversal, ...)

### Ongoing/future work

- certification in Isabelle/HOL
- support for continuous distributions
- support for concurrency
Questions?