Quantitative Separation Logic
Weakest Preexpectation Reasoning for Probabilistic Pointer Programs

Kevin Batz    Benjamin Kaminski
Joost-Pieter Katoen    Christoph Matheja    Thomas Noll

May 16th, Leiden
Why Probabilistic Pointer Programs?

Since 1976: randomized algorithms

Randomized skip list algorithms have the same asymptotic expected time bounds as balanced trees and are simpler, faster and use less space.[Pugh'89]

The expected running time of randomized splay trees is smaller than deterministic variants.[Albers & Karpinski'02]

More recently: approximate computing, artificial intelligence...

Lots of probabilistic programming languages (often without precise semantics)

Prominent example: Stan (10k+ active users)

wp-style verification of probabilistic pointer programs with separation logic

2 Christoph Matheja | Quantitative Separation Logic | Leiden 2019
Why Probabilistic Pointer Programs?

Since 1976: randomized algorithms
Why Probabilistic Pointer Programs?

Since 1976: randomized algorithms

“Randomized skip list algorithms have the same asymptotic expected time bounds as balanced trees and are simpler, faster and use less space.”

[Pugh’89]

More recently: approximate computing, artificial intelligence. . .

Lots of probabilistic programming languages (often without precise semantics)

Prominent example: Stan (10k+ active users)

wp-style verification of probabilistic pointer programs with separation logic
Why Probabilistic Pointer Programs?

Since 1976: randomized algorithms

“Randomized skip list algorithms have the same asymptotic expected time bounds as balanced trees and are **simpler, faster and use less space.**”

“**The expected running time** [of randomized splay trees] **is smaller** [than deterministic variants].”

[Pugh’89]  
[Albers & Karpinski’02]
Since 1976: randomized algorithms

“Randomized skip list algorithms have the same asymptotic expected time bounds as balanced trees and are simpler, faster and use less space.” [Pugh’89]

“The expected running time [of randomized splay trees] is smaller [than deterministic variants].” [Albers & Karpinski’02]

More recently: approximate computing, artificial intelligence...
Why Probabilistic Pointer Programs?

Since 1976: randomized algorithms

“Randomized skip list algorithms have the same asymptotic expected time bounds as balanced trees and are simpler, faster and use less space.”

“The expected running time [of randomized splay trees] is smaller [than deterministic variants].”

[Albers & Karpinski’02]

More recently: approximate computing, artificial intelligence. . .

Lots of probabilistic programming languages (often without precise semantics)
Why Probabilistic Pointer Programs?

Since 1976: randomized algorithms

"Randomized skip list algorithms have the same asymptotic expected time bounds as balanced trees and are simpler, faster and use less space." [Pugh'89]

“The expected running time [of randomized splay trees] is smaller [than deterministic variants].” [Albers & Karpinski'02]

More recently: approximate computing, artificial intelligence...

Lots of probabilistic programming languages (often without precise semantics)

Prominent example: Stan (10k+ active users)
Why Probabilistic Pointer Programs?

Since 1976: randomized algorithms

“Randomized skip list algorithms have the same asymptotic expected time bounds as balanced trees and are simpler, faster and use less space.”

“The expected running time of randomized splay trees is smaller than deterministic variants.”

More recently: approximate computing, artificial intelligence . . .

Lots of probabilistic programming languages (often without precise semantics)

Prominent example: Stan (10k+ active users)

wp-style verification of probabilistic pointer programs with separation logic
What are Probabilistic Programs?

• "Ordinary" programs with the additional ability to flip coins

• What does a probabilistic program $C$ do?
  - Run program $C$ on initial state $\sigma$
  - Obtain (sub-)distribution $J_{C, k, \sigma}$ over final states

• Operational semantics is a Markov chain

• What is the probability that a program behaves correctly?

• What is a program's expected behavior?

• What is its expected runtime?

[ESOP'16, JACM'18, ESOP'18]
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

\[
\{ \text{skip} \} \frac{1}{3} \{ x := x + 2 \}
\]

- What does a probabilistic program \( C \) do?
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 $\sigma$
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** \( [C]_\sigma \) 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 \( \llbracket C \rrbracket_\sigma \) 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** $[C]_{\sigma}$ over final states

- Operational semantics is a **Markov chain**
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

- Operational semantics is a **Markov chain**
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 \((\text{sub-})\text{distribution} \llbracket C \rrbracket_\sigma\) over final states

• Operational semantics is a Markov chain

• What is the probability that a program behaves correctly?

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

- “Ordinary” programs with the additional ability to flip coins

\[
\{ \text{	exttt{skip}} \}[1/3] \{ x := x + 2 \}
\]

- 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

- What is the probability that a program behaves correctly?

- What is a program’s expected behavior?
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 $\mathbb{P}_C,\sigma$ over final states

• Operational semantics is a Markov chain

• What is the probability that a program behaves correctly?

• What is a program’s expected behavior?

• What is its expected runtime? [ESOP’16, JACM’18, ESOP’18]
Probabilistic Program Phenomena

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

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

- does not always terminate
Probabilistic Program Phenomena

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

- does not always terminate
- terminates almost-surely
Probabilistic Program Phenomena

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

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

Proving that the expected runtime is \textbf{finite} is \textbf{strictly harder} than proving termination!

[Acta'18]
Probabilistic Program Phenomena

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

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

- does not always terminate
- terminates **almost-surely**
- expected runtime: constant
Probabilistic Program Phenomena

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

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

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

- does not always terminate
- terminates almost-surely
- expected runtime: constant
Probabilistic Program Phenomena

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

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

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

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

- does not always terminate
Probabilistic Program Phenomena

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

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

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

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

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

[Acta’18]

Christoph Matheja | Quantitative Separation Logic | Leiden 2019
Probabilistic Program Phenomena

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

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

- does not always terminate
- terminates almost-surely
- expected runtime: infinite
**Probabilistic Program Phenomena**

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

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

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

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

Proving that the expected runtime is finite is **strictly harder** than proving termination! [Acta’18]
Example: Array Randomization

```plaintext
randomize(array, n) {
    i := 0;
    while (0 ≤ i < n) {
        j := uniform(i, n - 1);
        swap(array, i, j);
        i := i + 1
    }
}
```
Goal: Formal Verification for Probabilistic Pointer Programs

Hoare logic

Preexpectations

Separation Logic
Goal: Formal Verification for Probabilistic Pointer Programs

- Hoare logic
- Preexpectations
- Separation Logic
- Quantitative Separation Logic
Verifying Quantitative Reliability for Programs That Execute on Unreliable Hardware

Michael Carbin, Sasa Misailovic, Martin C. Rinard

MIT CSAIL
{mcarbin, misailo, rinard}@mit.edu

Abstract
Emerging high-performance architectures are anticipated to contain unreliable components that may exhibit soft errors, which directly corrupt the results of computations. Full detection and masking of soft errors in challenging, expensive, and for some applications, inescapable (e.g., streaming, machine learning, and big data analytics) can often neither tolerate nor even mitigate 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 verified reliability engineering. The analysis takes a Rely program — a program 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 a quantitative reliability analysis on six computations implemented in Rely.

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 [29] and large [33] regions (which can execute acceptably and deliver acceptable performance) and in small [67] and large [33] regions (which must execute without error) and critical applications (which can execute acceptably and deliver acceptable performance) and in small [67] and large [33] regions (which must execute without error) and critical applications (which execute acceptably and deliver acceptable performance). A checkable computation can be augmented with an efﬁcient checker that certiﬁes the acceptability of the computation’s results [6, 9, 35, 55]. If the checker is efﬁcient, it allows the application to obtain an acceptable result without (or with at most selectively applied) mechanisms that detect and mask soft errors (such as voltage and energy efﬁcient execution that 2) delivers acceptably accurate results even though the number of fault-modes despite the presence of unmasked soft errors.

1.1 Background
Researchers have identiﬁed a range of both approximate...
Verifying Quantitative Reliability for Programs That Execute on Unreliable Hardware

Michael Carbin, Sasa Misailovic, Martin C. Rinard

MIT-CSAIL

{mcarbin, misailo, rinard}@mit.edu

Abstract

Emerging high-performance architectures are anticipated to contain unreliable components that may exhibit soft errors, which seriously corrupt the results of computations. Full detection and handling of soft errors in challenging, expensive, and long applications (e.g., streaming, machine learning, and big data analytics) can often tolerate random 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 node that a function produces.

We present a static quantitative reliability analysis that verifies quantitative requirements in the reliability of an application, enabling a developer to perform sound and trusted reliability engineering. This analysis takes a fully program-specified, quantitative reliability requirement, and returns a set of quantitative reliability requirements that characterizes the reliability of the underlying hardware components and ensures that the program satisfies the 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

1. Introduction

System reliability is a major challenge in emerging architectures. Energy-efficiency improvements are increasing major goals in design. However, aggressively pursuing these goals introduces the possibility of soft errors, which can seriously corrupt the results of computations. Full detection and handling of soft errors in challenging, expensive, and long applications (e.g., streaming, machine learning, and big data analytics) can often tolerate random soft errors.

Many computations, however, can tolerate random soft errors. An computation can often tolerate random soft errors. An approximate computation typically comes at the price of increased energy consumption, or both.

Many computations, however, can tolerate random soft errors. An approximate computation typically comes at the price of increased energy consumption, or both.

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 node that a function produces.

We present a static quantitative reliability analysis that verifies quantitative requirements in the reliability of an application, enabling a developer to perform sound and trusted reliability engineering. This analysis takes a fully program-specified, quantitative reliability requirement, and returns a set of quantitative reliability requirements that characterizes the reliability of the underlying hardware components and ensures that the program satisfies the 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

1. Introduction

System reliability is a major challenge in emerging architectures. Energy-efficiency improvements are increasing major goals in design. However, aggressively pursuing these goals introduces the possibility of soft errors, which can seriously corrupt the results of computations. Full detection and handling of soft errors in challenging, expensive, and long applications (e.g., streaming, machine learning, and big data analytics) can often tolerate random soft errors.

Many computations, however, can tolerate random soft errors. An approximate computation typically comes at the price of increased energy consumption, or both.

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 node that a function produces.

We present a static quantitative reliability analysis that verifies quantitative requirements in the reliability of an application, enabling a developer to perform sound and trusted reliability engineering. This analysis takes a fully program-specified, quantitative reliability requirement, and returns a set of quantitative reliability requirements that characterizes the reliability of the underlying hardware components and ensures that the program satisfies the 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

1. Introduction

System reliability is a major challenge in emerging architectures. Energy-efficiency improvements are increasing major goals in design. However, aggressively pursuing these goals introduces the possibility of soft errors, which can seriously corrupt the results of computations. Full detection and handling of soft errors in challenging, expensive, and long applications (e.g., streaming, machine learning, and big data analytics) can often tolerate random soft errors.

Many computations, however, can tolerate random soft errors. An approximate computation typically comes at the price of increased energy consumption, or both.
1. Introduction

System reliability is a major challenge for emerging architectures. Energy efficiency is becoming a major goal when designing large-scale computing systems. However, there is a need for approaches to enable 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 framework of decision–making based on partial state information spaces.

2. Current decision network frameworks cannot handle our applications, skip lists are a more natural representa-

3. Binary trees can be used for representing abstract data structures such as dictionaries and ordered lists. They work in a way to use arbitrary data types with minimal effort from developers to reason about the quantitative reliability of an application.

4. 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
bruttenberg, apfeffer@cra.com

Abstract
Existing decision-theoretic reasoning frameworks such as decision networks use simple data structures and processes. However, decision mechanisms 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

Within two use for representing abstract data structures such as dictionaries and ordered lists. They work in a way to use arbitrary data types with minimal effort from developers to reason about the quantitative reliability of an application. In particular, we retain the ability to use 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 defining 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 on standard probabilistic models are obtained for decision-theoretic models.
Quantitative Separation Logic (QSL)

“Instead of truth, we measure a quantity”
Quantitative Separation Logic (QSL)

“Instead of truth, we measure a quantity”

Predicate \( F \)

Expectation \( f \)
Quantitative Separation Logic (QSL)

“Instead of truth, we measure a quantity”

Predicate $F$  

Weakest precondition $\text{wp} \; [C] \; (F)$

Expectation $f$

Expected value $\int_{[C]} f$
Expectations
Expectations

\[ s : \text{Vars} \rightarrow \mathbb{Z} \]
Expectations

\[ s : \text{Vars} \rightarrow \mathbb{Z}, \quad h : \text{dom}(h) \rightarrow \mathbb{Z} \subseteq \mathbb{N}\setminus\{0\} \text{ finite} \]
Expectations

\[ \text{States} = \{(s, h) \mid s: \text{Vars} \to \mathbb{Z}, \ h: \ \underline{\text{dom}(h)} \to \mathbb{Z}\} \]

\( \subseteq \mathbb{N}\backslash\{0\} \) finite
$States = \{(s, h) \mid s : Vars \to \mathbb{Z}, \ h : \text{dom}(h) \to \mathbb{Z} \}$

Expectations: $f : States \to \mathbb{R}_{\geq 0}$
Expectations

\[
States = \{ (s, h) \mid s : Vars \to \mathbb{Z}, \quad h : \text{dom}(h) \to \mathbb{Z} \}
\]

\[\subseteq \mathbb{N}\setminus\{0\} \text{ finite}\]

Expectations: \[f : States \to \mathbb{R}_{\geq 0}\]
Think: random variable
Expectations

\[ States = \{(s, h) \mid s : Vars \to \mathbb{Z}, \ h : \text{dom}(h) \to \mathbb{Z}\} \]

Expectations: \( f : States \to \mathbb{R}_{\geq 0} \) Think: random variable

Examples:
Expectations

\[ States = \{(s, h) \mid s: Vars \rightarrow \mathbb{Z}, \; h: \text{dom}(h) \rightarrow \mathbb{Z}\} \]

**Expectations:** \( f: States \rightarrow \mathbb{R}_{\geq 0} \)  
Think: random variable

**Examples:**  
\[ x^2 = \lambda(s, h). s(x)^2 \]
Expectations

\[ \text{States} = \{ (s, h) \mid s: \text{Vars} \to \mathbb{Z}, \ h: \\overbrace{\text{dom}(h)}^{\subseteq \mathbb{N}\\backslash\{0\} \text{ finite}} \to \mathbb{Z} \} \]

\textbf{Expectations:} \quad f : \text{States} \to \mathbb{R}_\geq 0 \quad \text{Think: random variable}

\textbf{Examples:} \quad x^2 = \lambda(s, h). s(x)^2 \quad \text{size} = \lambda(s, h). |\text{dom}(h)|
Expectations

\[ \text{States} = \{(s, h) \mid s: \text{Vars} \to \mathbb{Z}, \quad h: \text{dom}(h) \to \mathbb{Z} \} \]

\[ \text{Expectations:} \quad f: \text{States} \to \mathbb{R}_{\geq 0} \quad \text{Think: random variable} \]

\[ \text{Examples:} \quad x^2 = \lambda(s, h). \ s(x)^2 \quad \text{size} = \lambda(s, h). \ |\text{dom}(h)| \]

\[ [\text{emp}] = \lambda(s, h). \begin{cases} 1 & \text{if } \text{dom}(h) = \emptyset \\ 0 & \text{otherwise} \end{cases} \]
Expectations

\[ \text{States} = \{(s, h) \mid s: \text{Vars} \to \mathbb{Z}, \ h: \underbrace{\text{dom}(h)}_{\subseteq \mathbb{N}\setminus\{0\} \text{ finite}} \to \mathbb{Z} \} \]

**Expectations:** \( f: \text{States} \to \mathbb{R}^\infty_{\geq 0} \) Think: random variable

**Examples:**

\[ x^2 = \lambda(s, h). s(x)^2 \]

\[ \text{size} = \lambda(s, h). |\text{dom}(h)| \]

\[ [\text{emp}] = \lambda(s, h). \begin{cases} 1 & \text{if } \text{dom}(h) = \emptyset \\ 0 & \text{otherwise} \end{cases} \]

\[ [x \mapsto y] = \lambda(s, h). \begin{cases} 1 & \text{if } s, h \models [x \mapsto y] \\ 0 & \text{otherwise} \end{cases} \]
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:

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

Notice:

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

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

Separating Conjunction
Separating Conjunction

Classical separating conjunction:

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

Classical separating conjunction:

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

Quantitative separating conjunction:

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

Classical separating implication:

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

Classical separating implication:

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

Quantitative separating implication:

\[f \rightarrow g = \lambda(s, h). \inf \left\{ \frac{g(s, h \ast h')}{f(s, h')} \right\} \text{ where } h' \perp h \text{ and } f(s, h') > 0 \text{ and not } f(s, h') = \infty = g(s, h \ast h')\]
Separating Implication

Classical separating implication:

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

Quantitative separating implication:

\[f \rightarrow g = \lambda(s, h). \inf \left\{ \frac{g(s, h \ast h')}{f(s, h')} \mid h' \perp h \text{ and } f(s, h') > 0 \text{ and not } f(s, h') = \infty = g(s, h \ast h') \right\}\]

Notice:

\[[F] \rightarrow g = \lambda(s, h). \inf \left\{ g(s, h \ast h') \mid h' \perp h \text{ and } (s, h') \models F \right\}\]
Properties of $\star$ and $\rightsquigarrow$ 

Classical adjointness:

$((F \star G) \implies J) \iff (F \implies (G \rightsquigarrow J))$
Properties of $\star$ and $\rightarrow$:

**Classical adjointness:**

\[(F \star G) \Rightarrow J \iff F \Rightarrow (G \star J)\]

**Quantitative Adjointness:**

\[f \star g \preceq j \text{ iff } f \preceq g \rightarrow j\]
Properties of $\star$ and $\rightarrow$

Classical adjointness:

$$((F \star G) \implies J) \iff (F \implies (G \rightarrow J))$$

Quantitative Adjointness:

$$f \star g \leq j \iff f \leq g \rightarrow j$$

+ virtually all properties of [Ishtiaq & O’Hearn’01, Reynolds’02]
The Weakest Preexpectation Transformer [Kozen’83, Mclver’99, Mclver & Morgan’05]

• Given: Probabilistic program $C$ and (post)expectation $f$
The Weakest Preexpectation Transformer [Kozen’83, Mclver’99, Mclver & Morgan’05]

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

Question: What is the expected value of $f$ after successful termination $C$?
The Weakest Preexpectation Transformer [Kozen’83, McIver’99, McIver & Morgan’05]

- **Given:** Probabilistic program $C$ and (post)expectation $f$
- **Question:** What is the expected value of $f$ after successful termination $C$?
The Weakest Preexpectation Transformer [Kozen’83, Mclver’99, Mclver & Morgan’05]

*Given:* Probabilistic program $C$ and (post)expectation $f$

*Question:* What is the expected value of $f$ after successful termination $C$?
The Weakest Preexpectation Transformer [Kozen’83, McIver’99, McIver & Morgan’05]

- **Given:** Probabilistic program \( C \) and (post)expectation \( f \)
- **Question:** What is the expected value of \( f \) after successful termination \( C \)?
The Weakest Preexpectation Transformer [Kozen’83, McIver’99, McIver & Morgan’05]

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

\[ \text{Exp} \left[ f(s_1, h_1) \cdot f(s_2, h_2) \cdot f(s_3, h_3) \ldots \right] \]
The Weakest Preexpectation Transformer [Kozen’83, Mclver’99, Mclver & Morgan’05]

- **Given**: Probabilistic program $C$ and (post)expectation $f$
- **Question**: What is the expected value of $f$ after successful termination $C$?
The Weakest Preexpectation Transformer [Kozen’83, Mclver’99, Mclver & Morgan’05]

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

- **Weakest preexpectation:** Mapping from initial state $(s, h)$ to expected value of $f$ evaluated in final states reached after successful termination of $C$ on $(s, h)$.

$$ \text{wp} ^{\llbracket C \rrbracket} (f) = \lambda(s, h). \int_{\llbracket C \rrbracket(s, h)} f \in \mathbb{E} = \{ f : \text{States} \to \mathbb{R}_{\geq 0} \} $$
Examples of Quantitative Specifications

**Postexpectation** $f$

$\lambda(s, h). 1$

**Weakest Preexpectation** $wp \llbracket c \rrbracket (f)$

Probability of memory-safe termination
Examples of Quantitative Specifications

<table>
<thead>
<tr>
<th>Postexpectation f</th>
<th>Weakest Preexpectation (<a href="f">c</a>)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\lambda(s, h). 1)</td>
<td>Probability of memory-safe termination</td>
</tr>
<tr>
<td>(</td>
<td>x</td>
</tr>
</tbody>
</table>
### Examples of Quantitative Specifications

<table>
<thead>
<tr>
<th>Postexpectation $f$</th>
<th>Weakest Preexpectation $\wp \llbracket c \rrbracket (f)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda(s, h). 1$</td>
<td>Probability of memory-safe termination</td>
</tr>
<tr>
<td>$</td>
<td>x</td>
</tr>
<tr>
<td>$\text{[emp]}$</td>
<td>Probability of terminating with an empty heap</td>
</tr>
</tbody>
</table>
## Examples of Quantitative Specifications

<table>
<thead>
<tr>
<th>Postexpectation $f$</th>
<th>Weakest Preexpectation $\wp , \llbracket \cdot \rrbracket (f)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda(s, h).1$</td>
<td>Probability of memory-safe termination</td>
</tr>
<tr>
<td>$</td>
<td>x</td>
</tr>
<tr>
<td>$\text{[emp]}$</td>
<td>Probability of terminating with an empty heap</td>
</tr>
<tr>
<td>$\text{len}(x, y)$</td>
<td>Expected length of list segment from $x$ to $y$</td>
</tr>
</tbody>
</table>
Weakest Preexpectation of Probabilistic Choice

\[
wp \left[ \{ C_1 \} \left[ p \right] \{ C_2 \} \right] (f) = p \cdot wp \left[ C_1 \right] (f) + (1 - p) \cdot wp \left[ C_2 \right] (f)
\]
Weakest Preexpectation of Memory Allocation

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

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

\[ s: \quad h: \]

\begin{array}{cccc}
  x & y & \cdots & \alpha \\
  u & w & \cdots & a \\
\end{array}

\begin{array}{cccc}
  \beta & \gamma & \cdots & b \\
  & c & \cdots & c \\
\end{array}
Weakest Preexpectation of Memory Allocation

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

\[
\begin{array}{cccc}
\text{s:} & x & y & \ldots & \alpha & \beta & \gamma & \ldots \\
& u & w & a & b & c & \ldots \\
\end{array}
\]

\[
\begin{array}{cccc}
\text{h:} & x & y & \ldots & \alpha & \beta & \gamma & \ldots \\
& a & b & c & \ldots \\
\end{array}
\]

\[
\begin{array}{cccc}
\text{h':} & x & y & \ldots & \alpha & \beta & \gamma & \ldots \\
& v & w & a & b & c & \ldots \\
\end{array}
\]

\[
\begin{array}{cccc}
\text{s'} & x & y & \ldots & \alpha & \beta & \gamma & \ldots \\
& v & w & a & b & c & \ldots \\
\end{array}
\]

\[ v \vdash s(e) \]
Weakest Preexpectation of Memory Allocation

\[
\begin{align*}
x & := \text{new}(e) \\
\{s[x/1], h \star \{1 \mapsto s(e)\}\} & \rightarrow \langle s \circ (x := \text{new}(e), s, h) \rangle
\end{align*}
\]
Weakest Preexpectation of Memory Allocation

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

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

\[ h : \begin{array}{c|c|c|c} x & y & \cdots & \alpha & \beta & \gamma & \cdots \\ \hline \end{array} \]

\[ s_1 : \begin{array}{c|c|c|c} x & y & \cdots & \alpha & \beta & \gamma & \cdots \\ \hline v & w & \cdots & a & b & c & \cdots \end{array} \]

\[ h_1 : \begin{array}{c|c|c|c} x & y & \cdots & \alpha & \beta & \gamma & \cdots \\ \hline \end{array} \]

\[ v : \begin{array}{c|c|c|c} x & y & \cdots & \alpha & \beta & \gamma & \cdots \\ \hline s(e) \end{array} \]

\[ v = 1 \rightarrow \langle s[x/1], h \star \{1 \mapsto s(e)\} \rangle \]

\[ v = 2 \rightarrow \langle s[x/2], h \star \{2 \mapsto s(e)\} \rangle \]
Weakest Preexpectation of Memory Allocation

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

\[ s: \]
\[ \begin{array}{cccc}
  x & y & \cdots & \\
  u & w & \cdots & \\
\end{array} \]

\[ h: \]
\[ \begin{array}{cccc}
  \alpha & \beta & \gamma & \\
  a & b & c & \\
\end{array} \]

\[ s_i: \]
\[ \begin{array}{cccc}
  x & y & \cdots & \\
  v & w & \cdots & \\
\end{array} \]

\[ h_i: \]
\[ \begin{array}{cccc}
  \alpha & \beta & \gamma & \\
  a & b & c & \\
\end{array} \]

\[ v: \]
\[ \begin{array}{c}
  * \\
  s(e) \\
\end{array} \]

\[ v = 1 \rightarrow \langle s[x/1], h \star \{1 \mapsto s(e)\} \rangle \]

\[ v = 2 \rightarrow \langle s[x/2], h \star \{2 \mapsto s(e)\} \rangle \]

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

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

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

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

\[ \langle s[x/1], h * \{1 \mapsto s(e)\} \rangle \]

\[ \langle s[x/2], h * \{2 \mapsto s(e)\} \rangle \]

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

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

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

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

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

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

\[
\begin{align*}
\inf_{v \in \mathbb{N}} [v \mapsto e] \rightarrow f[x/v]
\end{align*}
\]

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

\[
\begin{align*}
\langle x := \text{new}(e), s, h \rangle
\end{align*}
\]

\[
\begin{align*}
\langle s[x/1], h \star \{1 \mapsto s(e)\} \rangle
\end{align*}
\]

\[
\begin{align*}
\langle s[x/2], h \star \{2 \mapsto s(e)\} \rangle
\end{align*}
\]

\[
\begin{align*}
\vdash \text{infinite branching!}
\end{align*}
\]
### The Weakest Preexpectation Calculus for QSL

<table>
<thead>
<tr>
<th>C</th>
<th>wp ( [C] (f) )</th>
</tr>
</thead>
</table>

\[
\{ C_1 \} [p] \{ C_2 \} = p \cdot \wp \{ C_1 \} (f) + (1 - p) \cdot \wp \{ C_2 \} (f)
\]
The Weakest Preexpectation Calculus for QSL

\[
\begin{align*}
C & \quad \text{wp } \lceil C \rceil (f) \\
\{ C_1 \} [p] \{ C_2 \} & \quad p \cdot \text{wp } \lceil C_1 \rceil (f) + (1 - p) \cdot \text{wp } \lceil C_2 \rceil (f) \\
x & \quad := \text{new}(E) \\
x & \quad := \langle E \rangle \\
\langle E \rangle & \quad := E' \\
\text{free}(E) & \quad := \langle E \rangle \\
\end{align*}
\]
The Weakest Preexpectation Calculus for QSL

<table>
<thead>
<tr>
<th>C</th>
<th>( \text{wp} \left[ C \right] (f) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{skip}</td>
<td>( f )</td>
</tr>
<tr>
<td>( x := E )</td>
<td>( f \left[ x/E \right] )</td>
</tr>
<tr>
<td>( C_1; C_2 )</td>
<td>( \text{wp} \left[ C_1 \right] (\text{wp} \left[ C_2 \right] (f)) )</td>
</tr>
<tr>
<td>\text{if} (( B )) { ( C_1 ) } \text{else} { ( C_2 ) }</td>
<td>( [B] \cdot \text{wp} \left[ C_1 \right] (f) + [\neg B] \cdot \text{wp} \left[ C_2 \right] (f) )</td>
</tr>
<tr>
<td>\text{while} (( B )) { ( C' ) }</td>
<td>( \text{lfp} \ X. [\neg B] \cdot f + [B] \cdot \text{wp} \left[ C' \right] (X) )</td>
</tr>
<tr>
<td>{ ( C_1 ) } [( p )] { ( C_2 ) }</td>
<td>( p \cdot \text{wp} \left[ C_1 \right] (f) + (1 - p) \cdot \text{wp} \left[ C_2 \right] (f) )</td>
</tr>
<tr>
<td>( x := \text{new}(E) )</td>
<td>( \inf_{v \in \mathbb{N}} [v \mapsto E] \rightsquigarrow f \left[ x/v \right] )</td>
</tr>
<tr>
<td>( x := \langle E \rangle )</td>
<td>( \sup_{v \in \mathbb{Z}} [E \mapsto v] \rightsquigarrow ([E \mapsto v] \rightsquigarrow f \left[ x/v \right]) )</td>
</tr>
<tr>
<td>( \langle E \rangle := E' )</td>
<td>( [E \mapsto -] \rightsquigarrow ([E \mapsto E'] \rightsquigarrow f) )</td>
</tr>
<tr>
<td>free(( E ))</td>
<td>( [E \mapsto -] \rightsquigarrow f )</td>
</tr>
</tbody>
</table>
### The Weakest Preexpectation Calculus for QSL

<table>
<thead>
<tr>
<th>$C$</th>
<th>( \text{wp} \ [C] (f) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{skip}</td>
<td>( f )</td>
</tr>
<tr>
<td>( x := E )</td>
<td>( f [x/E] )</td>
</tr>
<tr>
<td>( C_1 ; C_2 )</td>
<td>( \text{wp} \ [C_1] (\text{wp} \ [C_2] (f)) )</td>
</tr>
<tr>
<td>\text{if} (( B )) { ( C_1 ) } \text{else} { ( C_2 ) }</td>
<td>( [B] \cdot \text{wp} \ [C_1] (f) + [-B] \cdot \text{wp} \ [C_2] (f) )</td>
</tr>
<tr>
<td>\text{while} (( B )) { ( C' ) }</td>
<td>( \text{lfp} \ X. \ [-B] \cdot f + [B] \cdot \text{wp} \ [C'] (X) )</td>
</tr>
<tr>
<td>{ ( C_1 ) } [p] { ( C_2 ) }</td>
<td>( p \cdot \text{wp} \ [C_1] (f) + (1 - p) \cdot \text{wp} \ [C_2] (f) )</td>
</tr>
<tr>
<td>( x := \text{new} (E) )</td>
<td>( \inf_{v \in \mathbb{N}} [v \mapsto E] \rightarrow f [x/v] )</td>
</tr>
<tr>
<td>( x := \langle E \rangle )</td>
<td>( \sup_{v \in \mathbb{Z}} [E \mapsto v] \star ([E \mapsto v] \rightarrow f [x/v]) )</td>
</tr>
<tr>
<td>( \langle E \rangle := E' )</td>
<td>( [E \mapsto -] \star ([E \mapsto E'] \rightarrow f) ) + probabilistic assignment</td>
</tr>
<tr>
<td>\text{free}(E)</td>
<td>( [E \mapsto -] \star f ) + procedures</td>
</tr>
</tbody>
</table>
QSL is a conservative extension of both SL and weakest preexpectations.
Theorem I: Conservativity

QSL is a conservative extension of both SL and weakest preexpectations.

\[ F, \ G \text{ SL formulas} \quad [F], [G] \text{ corresponding expectations} \]
Theorem I: Conservativity

QSL is a \textit{conservative extension of both SL and weakest preexpectations}.

\( F, G \) SL formulas \quad \llbracket F \rrbracket, \llbracket G \rrbracket \) corresponding expectations

For all states \((s, h)\) and all non-probabilistic pointer programs:

1) \( (s, h) \models F \iff \llbracket F \rrbracket(s, h) = 1 \)
Theorem I: Conservativity

**QSL is a conservative extension of both SL and weakest preexpectations.**

For all states \((s, h)\) and all non-probabilistic pointer programs:

1) \((s, h) \models F\) if and only if \([F](s, h) = 1\)

2) \{ F \} C \{ G \} valid if and only if \([F] \leq \text{wp} [C] ([G])\)
Theorem II: Soundness

For all programs $C$, expectations $f$ and states $(s, h)$,
Theorem II: Soundness

For all programs $C$, expectations $f$ and states $(s, h)$,
Theorem II: Soundness

For all programs $C$, expectations $f$ and states $(s, h)$,
Theorem II: Soundness

For all programs $C$, expectations $f$ and states $(s, h)$,

$$wp \begin{bmatrix} C \end{bmatrix} (f) (s, h) = \text{ExpRew}[f](⟨C, s, h⟩ \models □ \Diamond)$$

of operational Markov Decision Process.
Theorem III: The Quantitative Frame Rule

The classical frame rule:

For all SL formulas $F$, $G$, $R$ with $\text{Mod}(C) \cap \text{Vars}(R) = \emptyset$, 

\[
\frac{\{F\} \; C \; \{G\} }{ \{F \; \ast \; R\} \; C \; \{G \; \ast \; R\} }.
\]
Theorem III: The Quantitative Frame Rule

The classical frame rule:

For all SL formulas $F$, $G$, $R$ with $\text{Mod}(C) \cap \text{Vars}(R) = \emptyset$, we have:

$$\frac{\{ F \} \ C \ \{ G \}}{\{ F \star R \} \ C \ \{ G \star R \}}.$$

For $F = \text{wp} \left[ C \right] (G)$, this is equivalent to

$$\text{wp} \left[ C \right] (G) \star R \Rightarrow \text{wp} \left[ C \right] (G \star R).$$
Theorem III: The Quantitative Frame Rule

The classical frame rule:

For all SL formulas $F$, $G$, $R$ with $\text{Mod}(C) \cap \text{Vars}(R) = \emptyset$,

$$\frac{\{F\} \quad \{G\}}{\{F \star R\} \quad \{G \star R\}}.$$

For $F = \text{wp} \ [C] \ (G)$, this is equivalent to

$$\text{wp} \ [C] \ (G) \star R \Rightarrow \text{wp} \ [C] \ (G \star R).$$

The quantitative frame rule:

For all expectations $g$, $r$ with $\text{Mod}(C) \cap \text{Vars}(r) = \emptyset$,

$$\text{wp} \ [C] \ (g) \star r \leq \text{wp} \ [C] \ (g \star r).$$
A Small Example

\[\begin{align*}
    c &:= 1; \\
    \text{while}(c = 1) \{ \\
    \quad \{ c := 0 \} \left[\frac{1}{2}\right] \{ x := \text{new}(x) \} \\
    \}
\end{align*}\]
A Small Example

\[ c := 1; \]

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

\[
// [ls(x, 0)] \cdot \text{size}
\]
A Small Example

\[
c := 1;
\]

while (c = 1) {

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

// [ls(x, 0)] \cdot \text{size} + [c = 1]

\}

// [ls(x, 0)] \cdot \text{size}
A Small Example

c := 1;

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

    // [\text{ls}(x, 0)] \cdot \text{size}
A Small Example

\[ c := 1; \]

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

\[
\text{// } [\text{ls}(x, 0)] \cdot \text{size}
\]
\[ c := 1; \]

\[
\text{// } [c \neq 1] \cdot [\text{ls}(x, 0)] \cdot \text{size} + [c = 1] \cdot ([\text{ls}(x, 0)] \cdot \text{size} + \frac{1}{2}(1 + [c = 1]))
\]

\[
\text{while }(c = 1) \{
\text{// } [\text{ls}(x, 0)] \cdot \text{size} + \frac{1}{2}(1 + [c = 1])
\]

\[
\text{// } \frac{1}{2} \left( [\text{ls}(x, 0)] \cdot \text{size} + \inf_{\nu} [\nu \mapsto x] \rightarrow ([\text{ls}(x, 0)] \cdot \text{size} + [c = 1]) \right)
\]

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

\[
\text{// } [\text{ls}(x, 0)] \cdot \text{size} + [c = 1]
\]

\[
\text{// } [\text{ls}(x, 0)] \cdot \text{size}
\]
A Small Example

```plaintext
c := 1;
// [ls(x, 0)] \cdot \text{size} + [c = 1]
// [c \neq 1] \cdot [ls(x, 0)] \cdot \text{size} + [c = 1] \cdot ([ls(x, 0)] \cdot \text{size} + \frac{1}{2}(1 + [c = 1]))
while (c = 1) {
    // [ls(x, 0)] \cdot \text{size} + \frac{1}{2}(1 + [c = 1])
    // \frac{1}{2}\left([ls(x, 0)] \cdot \text{size} + \inf_{v \mapsto x} ([ls(x, 0)] \cdot \text{size} + [c = 1])\right)
    \{ c := 0 \} [1/2] \{ x := \text{new}(x) \}
    // [ls(x, 0)] \cdot \text{size} + [c = 1]
}
// [ls(x, 0)] \cdot \text{size}
```
A Small Example

```plaintext
// [ls(x, 0)] · size + 1
c := 1;
// [ls(x, 0)] · size + [c = 1]
// [c ≠ 1] · [ls(x, 0)] · size + [c = 1] · ([ls(x, 0)] · size + \( \frac{1}{2} (1 + [c = 1]) \))
while (c = 1) {
    // [ls(x, 0)] · size + \( \frac{1}{2} (1 + [c = 1]) \)
    // \( \frac{1}{2} ([ls(x, 0)] · size + \inf \{v \mapsto x\} \rightarrow ([ls(x, 0)] · size + [c = 1])) \)
    \{ c := 0 \} [1/2] \{ x := new(x) \}
    // [ls(x, 0)] · size + [c = 1]
}
// [ls(x, 0)] · size
```
Example I: Array Randomization

// $1/n! \cdot \max\{[\text{array} \mapsto \alpha_{\pi(0)}, \ldots, \alpha_{\pi(n-1)}] \mid \pi \text{ permutation}\}$

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

// [array $\mapsto \alpha_0, \ldots, \alpha_{n-1}$]
Example II: Faulty Garbage Collector

\[
\begin{align*}
// & \quad [\text{tree}(x)] \cdot (1 - p)^{\text{size}} \\
\text{delete}(x)\{ \\
\quad \text{if } (x \neq 0) \{ & \quad \text{fails with probability } p \\
\quad \quad \{ \text{skip} \} [p] \{ \\
\quad \quad \quad l := \langle x \rangle; r := \langle x + 1 \rangle; \\
\quad \quad \quad \text{delete}(l); \\
\quad \quad \quad \text{delete}(r); \\
\quad \quad \quad \text{free}(x); \text{free}(x + 1) \\
\quad \quad \} \} \\
\} \\
// & \quad [\text{emp}]  
\end{align*}
\]
Example III: Randomized Meldable Heaps

// [root ≠ 0] · [tree(root)] · log(1 + 1/2 · size)
randomLeaf(root) {
    nextL := ⟨root⟩;
    nextR := ⟨root + 1⟩;
    if (nextL = 0 and nextR = 0) {
        return root
    } else {
        { next := nextL } [0.5] { next := nextR };
        return randomLeaf(root)
    }
} // [tree(root)] · (1 * [path(root, result)] · 1/2 · size)
Example IV: Lossy List Reversal

// 0.5 · [hd ≠ 0] · len(hd, 0)
lossyReversal(hd) {
    r := 0;
    while (hd ≠ 0) {
        t := ⟨hd⟩;
        ⟨hd⟩ := r;
        [0.5] free(hd);
        r := hd
        hd := t
    }
}  // len(r, 0)
Conclusion

Quantitative Separation Logic... 

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

Further Reading


Future Work

- Mechanization / Automation
- Support for continuous distributions

Thank you for listening!
Conclusion

Quantitative Separation Logic . . .

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

Further Reading


Thank you for listening!
Conclusion

Quantitative Separation Logic... 
• as an assertion language
• as a verification system
• as a conservative, sound extension of separation logic

Further Reading


Future Work
• Mechanization / Automation
• Support for continuous distributions
Conclusion

Quantitative Separation Logic... 
- as an assertion language
- as a verification system
- as a conservative, sound extension of separation logic

Further Reading


Future Work
- Mechanization / Automation
- Support for continuous distributions

Thank you for listening!