Quantitative Separation Logic
A Logic for Reasoning about Probabilistic Pointer Programs

Kevin Batz    Benjamin Lucien Kaminski
Joost-Pieter Katoen    Christoph Matheja¹    Thomas Noll

POPL 2019

January 16th, Cascais/Lisbon, Portugal

¹Supported by DFG Grant NO 401/2-1
Goal: Formal Verification for Probabilistic Pointer Programs
Goal: Formal Verification for Probabilistic Pointer Programs

Hoare logic
Goal: Formal Verification for Probabilistic Pointer Programs

Hoare logic

The world has pointers in it
“[The null reference] has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.”
The world has pointers in it

“[The null reference] has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.”

**Other problems:** aliasing, sharing between data structures, . . .
The world has pointers in it

"[The null reference] has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years."

Other problems: aliasing, sharing between data structures, ... 

Separation logic enables compositional reasoning about pointer programs

Christoph Matheja | Quantitative Separation Logic | POPL 2019
Goal: Formal Verification for Probabilistic Pointer Programs

The world has pointers in it

Hoare logic

separation logic
Goal: Formal Verification for Probabilistic Pointer Programs

Hoare logic

The world has pointers in it

separation logic

The world is quantitative

2 Christoph Matheja | Quantitative Separation Logic | POPL 2019
“the Boolean partition of software into correct and incorrect programs falls short of the practical need to assess the behavior of software in a more nuanced fashion.”
"the Boolean partition of software into correct and incorrect programs falls short of the practical need to assess the behavior of software in a more nuanced fashion."

**Examples:** failure probabilities, resource consumption, expected runtimes, ...
The world is quantitative

“the Boolean partition of software into correct and incorrect programs falls short of the practical need to assess the behavior of software in a more nuanced fashion.”

Examples: failure probabilities, resource consumption, expected runtimes, . . .

Weakest preexpectations enable compositional reasoning about quantitative properties of probabilistic programs
Goal: Formal Verification for Probabilistic Pointer Programs

- Hoare logic
- Separation logic
- Preexpectations

The world has pointers in it
The world is quantitative
Goal: Formal Verification for Probabilistic Pointer Programs

The world has pointers in it

separation logic

Hoare logic

The world is quantitative

preexpectations

The world is quantitative
Goal: Formal Verification for Probabilistic Pointer Programs

The world has pointers in it

Hoare logic

The world is quantitative

separation logic

preexpectations

The world is quantitative

The world has pointers in it
Goal: Formal Verification for Probabilistic Pointer Programs

The world has pointers in it

Hoare logic

The world is quantitative

preexpectations

The world is quantitative

quantitative separation logic

The world has pointers in it

separation logic
Verifying Quantitative Reliability for Programs That Execute on Unreliable Hardware

Michael Carbin Sasa Misailovic Martin C. Rinard

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

Abstract

Emerging high-performance architectures are anticipated to contain unreliable components that may exhibit soft errors, which identify corrupt the results of computations. Full detection and masking of soft errors is challenging, expensive, and, for some applications, infeasible. For example, systems that support approximate computations (including multimedia, machine learning, and big data analytics) can often tolerate occasional 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. This analysis takes a rely program as input, checks that the program satisfies its reliability specification when executed on the underlying unreliable hardware platform, and, for some applications, unnecessary. For example, approximate computations (such as multimedia programs) can often tolerate occasional soft errors.

Key Contributions

1. Rely, a programming language that enables developers to reason about the quantitative reliability of an application.
2. A sound and verified reliability analysis that checks that the program satisfies its reliability specification when executed on the underlying unreliable hardware platform.
3. A tool for quantitatively checking reliability requirements.

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 [23] and large [67] components and even in large [67]. A fixed component that contains unreliable components that may exhibit 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. This analysis takes a rely program as input, checks that the program satisfies its reliability specification when executed on the underlying unreliable hardware platform, and, for some applications, unnecessary. For example, approximate computations (such as multimedia programs) can often tolerate occasional soft errors.

Key Contributions

1. Rely, a programming language that enables developers to reason about the quantitative reliability of an application.
2. A sound and verified reliability analysis that checks that the program satisfies its reliability specification when executed on the underlying unreliable hardware platform.
3. A tool for quantitatively checking reliability requirements.

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, misailovic, rinard}@mit.edu

Abstract

Emerging high-performance architectures are anticipated to contain unreliable components that may exhibit soft errors, which identify corrupt the results of computations. Full detection and masking of soft errors is challenging, expensive, and, for some applications, infeasible. For example, applications that require high-speed execution (e.g., multimedia, machine learning, and big data analytics) can often tolerate some 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 such nodes that a function produces.

We present a static quantitative reliability analysis that verifies quantitative requirements with the reliability of an application, enabling a developer to perform sound and fault-tolerant reliability engineering. The analysis takes a Rely program and produces an annotated program that certifies the reliability of the underlying hardware component and that the program satisfies the reliability requirements when executed on the underlying hardware platform. We demonstrate the application of quantitative reliability analysis on six computations implemented in Rely.

Categories and Subject Descriptors: F.1.1 [Logics and Meanings of Programs]: Specifying and Verifying Programs

1. Introduction

System reliability is a major challenge for emerging architectures. Energy efficiency in these systems is becoming more critical, and soft errors are frequently initiated. For example, both software and hardware errors are prevalent due to the use of unreliable components.

Many researchers have proposed detection and masking techniques for detecting and resolving soft errors [33, 34], but these approaches often introduce a trade-off between reliability and performance.

Full data structures are data structures that use probabilistic balancing rather than strictly ordered balancing. As a result, the algorithms for insertion and deletion in skip lists are much simpler and significantly faster than equivalent algorithms for balanced trees.

Skip lists are data structures that use probabilistic balancing rather than strictly ordered balancing. As a result, the algorithms for insertion and deletion in skip lists are much simpler and significantly faster than equivalent algorithms for balanced trees.

William Pugh

1988 can be used for representing abstract data types such as dictionaries and ordered lists. These work well when the elements are inserted in a random order. Linear sequences of elements, such as inserting the elements in order, provide a simple implementation and a worst-case optimal time, but inserting elements in a random order is not practical.

In this paper, we describe a self-adjusting tree algorithm, skip lists, that allows developers to specify the reliability requirements for such nodes that a function produces.

We present a static quantitative reliability analysis that verifies quantitative requirements with the reliability of an application, enabling a developer to perform sound and fault-tolerant reliability engineering. The analysis takes a Rely program and produces an annotated program that certifies the reliability of the underlying hardware component and that the program satisfies the reliability requirements when executed on the underlying hardware platform. We demonstrate the application of quantitative reliability analysis on six computations implemented in Rely.

Categories and Subject Descriptors: F.1.1 [Logics and Meanings of Programs]: Specifying and Verifying Programs

[OOPSLA’13] [CACM’90]
Verifying Quantitative Reliability for Programs That Execute on Unreliable Hardware
Michael Carbin, Sasa Misailović, Martin C. Rinard
MIT CSAIL
{mcarbin, misailo, rinard}@csail.mit.edu

Abstract
Emerging high-performance architectures are anticipated to contain untrusted components that may exhibit soft errors, which severely corrupt the results of computations. Full detection and masking of soft errors is challenging, expensive, and, for some applications, unnecessary. For example, approximations to program results (e.g., rounding, machine-learning, and big data analysis) can often tolerate some amount of soft error.

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 code path 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 reliable reliability engineering. The analysis takes a rely program as input and produces output that characterizes the reliability of the underlying hardware components and ensures that the program satisfies its reliability requirements 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 [Languages and Aspect of Programming]: Specify and Verifying and Reasoning about Programs

1 Introduction
System reliability is a major challenge for emerging architectures. Energy efficiency is becoming ever more important. How can we reduce the overall cost of processing, machine learning, and big data analysis? One common strategy is to detect and mask soft errors by runs of data processing. Even if many iterations are required to converge to the correct solution, increased adaptability and increased fault tolerance make the computation worthwhile.

1.1 Background
Researchers have identified a range of techniques to reduce the reliability of the computation's results [8, 9, 35, 55]. If the checker were completely reliable, we could simply perform the computations normally and simply check the result. The checker might be less reliable than the computation, however. In this case, the checker's result is not enough; we must perform multiple computations and then check their results for consistency.

1.2 Our Contributions
We present Rely, a programming language and analysis tool for specifying and verifying quantitative reliability for programs that execute on unreliable hardware. Our tool does not require storing any auxiliary information within the program code itself.

1.3 Related Work
Many computations, however, can tolerate occasional approximate results often enough to satisfy the needs of their users. Many multimedia, financial, machine learning, and big data analytics applications (e.g., weather predictions, financial market forecasts, machine learning, and big data analysis) can often acceptably tolerate occasional approximate results.

1.4 Paper Organization
We present the design and implementation of Rely in Section 2. In Section 3, we present our static quantitative reliability analysis, and in Section 4, we analyze six Rely programs. In Section 5, we conclude with a discussion of related work.

2 Rely: A Programming Language for Reliability Engineering
Rely is a general-purpose programming language that allows developers to specify the reliability requirements for each code path that a function produces. Rely is a statically typed, imperative language with strong typing and garbage collection. It supports a variety of programming constructs, including if statements, loops, and functions.

2.1 Syntax
Rely programs are written in a simple, readable syntax. A Rely program consists of a main function, which is executed when the program is run.

2.2 Semantics
Rely programs are interpreted by a compiler that generates code for a virtual machine. The virtual machine is a simple, stack-based machine that supports basic arithmetic operations, comparisons, and boolean operations.

2.3 Static Analysis
Rely includes a static analysis tool that verifies the quantitative reliability of a program. The analysis tool takes a Rely program as input and produces output that characterizes the reliability of the program's reliability requirements.

3 Static Quantitative Reliability Analysis
Rely includes a static quantitative reliability analysis tool that checks the reliability of a program. The analysis tool takes a Rely program as input and produces output that characterizes the reliability of the program. The output is a set of quantitative reliability requirements for each code path that a function produces.

3.1 Reliability Requirements
Rely supports a variety of quantitative reliability requirements, including the probability that a function produces the correct result when executed on unreliable hardware. Rely allows developers to specify these requirements for each code path that a function produces.

3.2 Verification
The static analysis tool checks the quantitative reliability requirements for each code path that a function produces. The tool verifies that the program satisfies its reliability requirements when executed on unreliable hardware.

4 Case Studies
We present six case studies of Rely programs. The case studies include a compiler, a database, and a machine-learning program.

4.1 Compiler
We present a compiler that generates code for a virtual machine. The compiler takes a Rely program as input and produces code for the virtual machine.

4.2 Database
We present a database that stores and retrieves data. The database is implemented as a distributed system that stores data on unreliable hardware.

4.3 Machine Learning
We present a machine-learning program that performs data analysis. The program takes a Rely program as input and produces output that characterizes the reliability of the program's reliability requirements.

5 Conclusion
We present Rely, a programming language and analysis tool for specifying and verifying quantitative reliability for programs that execute on unreliable hardware. Rely is a general-purpose programming language with strong typing and garbage collection. It supports a variety of programming constructs, including if statements, loops, and functions.

5.1 Future Work
We plan to extend Rely to support more advanced programming constructs, such as concurrency and higher-order functions. We also plan to extend the static analysis tool to support more advanced reliability requirements.
Quantitative Separation Logic (QSL)

Assertion language

Verification system

Theorems

Examples
Assertion Language: States and Expectations

States = \{(s, h) | s : \text{Vars} \rightarrow \mathbb{Z}, h : \text{dom}(h) \subseteq \mathbb{N} \setminus \{0\} \text{ finite} \rightarrow \mathbb{Z}\}
Assertion Language: States and Expectations

\[ s : Vars \rightarrow \mathbb{Z} \]
Assertion Language: States and Expectations

\[ s : \text{Vars} \rightarrow \mathbb{Z}, \quad h : \text{dom}(h) \rightarrow \mathbb{Z} \]
\[ \subseteq \mathbb{N}\backslash\{0\} \text{ finite} \]
Assertion Language: States and Expectations

\[
States = \{(s, h) \mid s : \text{Vars} \rightarrow \mathbb{Z}, \quad h: \underline{\text{dom} (h)} \rightarrow \mathbb{Z} \} \\
\subset \mathbb{N}\{0\} \text{ finite}
\]
Assertion Language: States and Expectations

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

Expectations: \[ f: States \rightarrow \mathbb{R}_\geq 0 \]
States = \{(s, h) \mid s \colon \text{Vars} \to \mathbb{Z}, \quad h \colon \text{dom}(h) \to \mathbb{Z} \}
\subseteq \mathbb{N} \setminus \{0\} \text{ finite}

Expectations: \quad f : States \to \mathbb{R}_{\geq 0}^\infty

Examples: \quad x^2 = \lambda(s, h). s(x)^2
Assertion Language: States and Expectations

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

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

**Examples:**

\[
x^2 = \lambda(s, h). s(x)^2 \quad \text{size} = \lambda(s, h). |\text{dom}(h)|
\]
Assertion Language: States and Expectations

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

Expectations: \[ f: States \rightarrow \mathbb{R}_{\geq 0} \]

Examples:

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

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

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

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

Expectations:

\[ f: \text{States} \rightarrow \mathbb{R}_{\geq 0} \]

Examples:

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

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

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

Classical conjunction: \( \varphi \land \psi \)
Assertion Language: Separating Conjunction

**Classical conjunction:** \( \varphi \land \psi \)  
**Quantitative conjunction:** \( f \cdot g = \lambda(s, h). f(s, h) \cdot g(s, h) \)
Assertion Language: Separating Conjunction

Classical conjunction: $\phi \land \psi$  
Quantitative conjunction: $f \cdot g = \lambda(s, h). f(s, h) \cdot g(s, h)$

Classical separating conjunction:

$$(s, h) \models \phi \star \psi$$  
iff  
$\exists h_1, h_2: h = h_1 \star h_2$  
and  
$$(s, h_1) \models \phi$$  
and  
$$(s, h_2) \models \psi$$
Assertion Language: Separating Conjunction

Classical conjunction: $\varphi \land \psi$  
Quantitative conjunction: $f \cdot g = \lambda(s, h). f(s, h) \cdot g(s, h)$

Classical separating conjunction:

$$(s, h) \models \varphi \star \psi \iff \exists h_1, h_2: h = h_1 \star h_2 \text{ and } (s, h_1) \models \varphi \text{ and } (s, h_2) \models \psi$$

\[
\begin{array}{ccc}
1: & 2: & 3: \\
\ast & a & b & c
\end{array}
\begin{array}{ccc}
1: & 2: & 3: \\
\ast & a & b & c
\end{array}
\begin{array}{ccc}
1: & 2: & 3: \\
\ast & a & b & c
\end{array}
\begin{array}{ccc}
1: & 2: & 3: \\
\ast & a & b & c
\end{array}
\begin{array}{ccc}
1: & 2: & 3: \\
\ast & a & b & c
\end{array}
\begin{array}{ccc}
1: & 2: & 3: \\
\ast & a & b & c
\end{array}\ldots
\]
Assertion Language: Separating Conjunction

**Classical conjunction:** $\varphi \land \psi$  
**Quantitative conjunction:** $f \cdot g = \lambda(s, h). f(s, h) \cdot g(s, h)$

**Classical separating conjunction:**

$$(s, h) \models \varphi \star \psi \quad \text{iff} \quad \exists h_1, h_2: h = h_1 \star h_2 \quad \text{and} \quad (s, h_1) \models \varphi \quad \text{and} \quad (s, h_2) \models \psi$$

**Quantitative separating conjunction:**

$$f \star g = \lambda(s, h). \max \{ f(s, h_1) \cdot g(s, h_2) \mid h = h_1 \star h_2 \}$$
Assertion Language: Separating Implication

⋆ and ⋿ are adjoint:  \((\varphi \star \psi) \Rightarrow \rho\) iff \(\varphi \Rightarrow (\psi \to \rho)\)
Assertion Language: Separating Implication

⋆ and →⋆ are adjoint: $$(\varphi \star \psi) \Rightarrow \rho \iff \varphi \Rightarrow (\psi \rightarrow^\star \rho)$$

Classical separating implication:

$$(s, h) \models \varphi \rightarrow^\star \psi \iff \forall h' \text{ with } \text{dom}(h) \cap \text{dom}(h') = \emptyset \text{ and } (s, h') \models \varphi: (s, h \star h') \models \psi$$
\begin{itemize}
\item \textbf{and} \textbullet \textbf{and} \textbullet \textbf{are adjoint:} \quad (\varphi \textbullet \psi) \Rightarrow \rho \iff \varphi \Rightarrow (\psi \textbullet \rho)
\end{itemize}

**Classical separating implication:**

\[(s, h) \models \varphi \textbullet \psi \iff \forall h' \text{ with } \text{dom}(h) \cap \text{dom}(h') = \emptyset \text{ and } (s, h') \models (s, h \textbullet h') \models \psi\]

\begin{itemize}
\item \textbf{and} \textbullet \textbf{are adjoint:} \quad (f \textbullet g) \leq r \iff f \leq (g \rightarrow r)
\end{itemize}
Assertion Language: Separating Implication

⋆ and →⋆ are adjoint: \((\varphi \star \psi) \Rightarrow \rho \iff \varphi \Rightarrow (\psi \rightarrow \rho)\)

Classical separating implication:

\((s, h) \models \varphi \rightarrow \star \psi \iff \forall h' \text{ with } \text{dom}(h) \cap \text{dom}(h') = \emptyset \text{ and } (s, h') \models \varphi: (s, h \star h') \models \psi\)

⋆ and →⋆ are adjoint: \((f \star g) \leq r \iff f \leq (g \rightarrow \star r)\)

Quantitative separating implication:

\([\varphi] \rightarrow \star g = \lambda(s, h). \inf \{g(s, h \star h') \mid \text{dom}(h) \cap \text{dom}(h') = \emptyset \text{ and } [\varphi](s, h') = 1\}\)
wp [] : Expectations → Expectations
wp [c] : \textit{Expectations} \rightarrow \textit{Expectations}

\begin{align*}
\begin{array}{c}
\text{c} \\
\text{f}
\end{array}
\end{align*}

\textit{f} is a \textbf{postexpectation} evaluated in \textit{final states}.
Verification System: The Weakest Preexpectation Transformer

\[ wp \ [c] : \text{Expectations} \rightarrow \text{Expectations} \]

\[ wp \ [c] (f) \]

\[ c \]

\[ f \]

\( wp \ [c] (f) \) is an expectation mapping an initial state \((s, h)\) to the expected value of \(f\) after successful termination of \(c\) on \((s, h)\).

\( f \) is a postexpectation evaluated in final states.
Verification System: Examples of Quantitative Specifications

**Postexpectation $f$**

\[ \lambda(s, h). 1 \]

**$wp \lfloor c \rfloor (f)$**

Probability of memory-safe termination
<table>
<thead>
<tr>
<th>Postexpectation $f$</th>
<th>$\text{wp} \left[ c \right] (f)$</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>
</tbody>
</table>
### Verification System: Examples of Quantitative Specifications

<table>
<thead>
<tr>
<th>Postexpectation $f$</th>
<th>$\text{wp } [c] (f)$</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>
### Verification System: Examples of Quantitative Specifications

<table>
<thead>
<tr>
<th>Postexpectation ( f )</th>
<th>( \text{wp} \left[ [c] \right] (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>
Verification System: Weakest Preexpectation of Memory Allocation

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

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

\[
\begin{array}{cccc}
  s: & h: \\
  \begin{array}{c}
x: \ y: \ \cdots \\
u \ w \ \cdots \\
\alpha: \ \beta: \ \gamma: \ \cdots \\
a \ b \ c \ \cdots \\
\end{array}
\end{array}
\]
Verification System: Weakest Preexpectation of Memory Allocation

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

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

\[
\begin{array}{c|c|c|c}
 s_\downarrow & h_\downarrow \\
\hline
 x & y & \ldots & \alpha & \beta & \gamma & \ldots \\
 v & w & \ldots & a & b & c & \ldots \\
\end{array}
\]

\[
\overset{*}{v} : s(e)
\]
Verification System: Weakest Preexpectation of Memory Allocation

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

\[ s : \]
\[ x: y: \quad \alpha: \beta: \gamma: \quad u: w: \quad a: b: c: \quad \ldots \]

\[ h : \]
\[ x: y: \quad \alpha: \beta: \gamma: \quad u: w: \quad a: b: c: \quad \ldots \]

\[ s_1 : \]
\[ x: y: \quad \alpha: \beta: \gamma: \quad v: w: \quad a: b: c: \quad \ldots \]

\[ h_1 : \]
\[ v: \]
\[ s(e) \]

\[ v = 1 \]
\[ \langle x := \text{new}(e), s, h \rangle \]

\[ \langle s[x/1], h \star \{1 \mapsto s(e)\} \rangle \]
Verification System: Weakest Preexpectation of Memory Allocation

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

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

\[
\begin{array}{c}
\text{s\_w:} \\
x: y: \ldots \\
v \quad w
\end{array} 
\quad \text{h\_w:} 
\begin{array}{c}
x: y: \ldots \\
a \quad b \quad c \quad \ldots
\end{array}
\]

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

\[ v = 2 \quad \langle x := \text{new}(e), s, h \rangle \quad \langle s[x/2], h \star \{2 \mapsto s(e)\} \rangle \]
Verification System: Weakest Preexpectation of Memory Allocation

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

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

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

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

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

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

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

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

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

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

\[
\langle x := \text{new}(e), s, h \rangle
\quad\quad
\langle s[x/1], h \ {1} \mapsto s(e) \rangle
\quad\quad
\langle s[x/2], h \ {2} \mapsto s(e) \rangle
\]

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

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

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

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

\[ v = 1 \]

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

\[ v = 2 \]

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

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

\[ \inf_{v \in \mathbb{Z}} [v \mapsto e] \rightarrow f[x/v] \]

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

\[ \langle x := \text{new}(e), s, h \rangle \quad v = 1 \quad \langle s[x/1], h \uplus \{1 \mapsto s(e)\} \rangle \]

\[ \langle x := \text{new}(e), s, h \rangle \quad v = 2 \quad \langle s[x/2], h \uplus \{2 \mapsto s(e)\} \rangle \]

\[ \vdash \text{infinite branching!} \]
Verification System: The Weakest Preexpectation Calculus for QSL

\[ c \quad \text{wp} \ [c] (f) \]

\[ x := \text{new} (e) \quad \text{inf} \ [\nu \mapsto e] \rightarrow f [x/\nu] \]
<table>
<thead>
<tr>
<th>$c$</th>
<th>$\text{wp } [c] (f)$</th>
</tr>
</thead>
</table>

\[
\begin{align*}
\{ c_1 \} [p] \{ c_2 \} & \quad p \cdot \text{wp } [c_1] (f) + (1 - p) \cdot \text{wp } [c_2] (f) \\
\text{\texttt{x := new}} (e) & \quad \inf \limits_{v \in \mathbb{Z}} [v \mapsto e] \mapsto f [x/v]
\end{align*}
\]
```
<table>
<thead>
<tr>
<th>c</th>
<th>\text{wp} [c] (f)</th>
<th>Do not read this!</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{skip}</td>
<td>f</td>
<td></td>
</tr>
<tr>
<td>\text{x} := e</td>
<td>f [x/e]</td>
<td></td>
</tr>
<tr>
<td>\text{c}_1 \circ \text{c}_2</td>
<td>\text{wp} [c_1] \left( \text{wp} [c_2] (f) \right)</td>
<td></td>
</tr>
<tr>
<td>\text{if} (b) { \text{c}_1 } \text{else} { \text{c}_2 }</td>
<td>[b] \cdot \text{wp} [c_1] (f) + [\neg b] \cdot \text{wp} [c_2] (f)</td>
<td></td>
</tr>
<tr>
<td>\text{while} (b) { c' }</td>
<td>\text{lfp} g. \left[ \neg b \right] \cdot f + [b] \cdot \text{wp} [c'] (g)</td>
<td></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>
<td></td>
</tr>
<tr>
<td>\text{x} := \text{new} (e)</td>
<td>\text{inf} _{v \in Z} [v \mapsto e] \mapsto f [x/v]</td>
<td></td>
</tr>
<tr>
<td>\text{x} := *e</td>
<td>\text{sup} _{v \in Z} [e \mapsto v] \mapsto (e \mapsto v) \mapsto f [x/v]</td>
<td></td>
</tr>
<tr>
<td>*e := e'</td>
<td>[e \mapsto -] \mapsto (e \mapsto e') \mapsto f</td>
<td>+ \text{probabilistic assignment}</td>
</tr>
<tr>
<td>\text{free} (e)</td>
<td>[e \mapsto -] \mapsto f</td>
<td>+ \text{procedures}</td>
</tr>
</tbody>
</table>
```
Theorem I: Conservativity

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.

\( \phi, \psi \) SL formulas \( [\phi], [\psi] \) corresponding expectations
Theorem I: Conservativity

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

**ϕ, ψ** SL formulas  \( [\varphi], [\psi] \) corresponding expectations

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

1) \((s, h) |= \varphi \) iff \( [\varphi](s, h) = 1 \)
Theorem I: Conservativity

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

\( \varphi, \psi \) SL formulas \( \llbracket \varphi \rrbracket, \llbracket \psi \rrbracket \) corresponding expectations

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

1) \((s, h) \models \varphi \) iff \( \llbracket \varphi \rrbracket (s, h) = 1 \)

2) \( \{ \varphi \} c \{ \psi \} \) valid iff \( \llbracket \varphi \rrbracket \leq \text{wp} \llbracket c \rrbracket (\llbracket \psi \rrbracket) \)
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 $(s, h)$,

$$\text{wp} \ [c] (f) (s, h) = \text{ExpRew}[f](\langle c, s, h \rangle \models \diamond \lozenge).$$

of operational MDP
Theorem III: The Quantitative Frame Rule

The classical frame rule:

For all SL formulas \( \varphi, \psi, \rho \) with \( \text{Mod}(c) \cap \text{Vars}(\rho) = \emptyset \),

\[
\frac{\{ \varphi \} \ c \ \{ \psi \}}{\{ \varphi \star \rho \} \ c \ \{ \psi \star \rho \}}.
\]
Theorem III: The Quantitative Frame Rule

The classical frame rule:

For all SL formulas \( \varphi, \psi, \rho \) with \( \text{Mod}(c) \cap \text{Vars}(\rho) = \emptyset \),

\[
\begin{align*}
\{ \varphi \} \ c \ & \ {\psi} \ \Rightarrow \ \\[ \varphi \star \rho \} \ c \ & \ {\psi \star \rho} \ \\
\end{align*}
\]

For \( \varphi = \text{wp} \left[ c \right] (\psi) \), this is equivalent to

\[
\text{wp} \left[ c \right] (\psi \star \rho) \Rightarrow \text{wp} \left[ c \right] (\psi \star \rho) \ \\
\]
Theorem III: The Quantitative Frame Rule

The classical frame rule:

For all SL formulas $\varphi, \psi, \rho$ with $\text{Mod}(c) \cap \text{Vars}(\rho) = \emptyset$,

$$\{ \varphi \} c \{ \psi \} = \{ \varphi * \rho \} c \{ \psi * \rho \}.$$

For $\varphi = \text{wp} \left[ c \right] \left( \psi \right)$, this is equivalent to

$$\text{wp} \left[ c \right] \left( \psi * \rho \right) \Rightarrow \text{wp} \left[ c \right] \left( \psi * \rho \right).$$

The quantitative frame rule:

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

$$\text{wp} \left[ c \right] \left( f * g \right) \leq \text{wp} \left[ c \right] \left( f * g \right).$$
Example I: Array Randomization

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

The probability of computing any fixed permutation of array is at most \( \frac{1}{n!} \).

\[
\text{wp} \left[ \text{randomize}(\text{array}, n) \right] ([\text{array} \mapsto \alpha_0, \ldots, \alpha_{n-1}]) \leq \frac{1}{n!}
\]
Example II: Faulty Garbage Collector

```
delete(x) {
    if (x \neq 0) { // fails with probability \( p \)
        { skip } [p] {
            l := \ast x ; r := \ast (x + 1) ;
            delete(l) ;
            delete(r) ;
            free(x) ; free(x + 1)
        } else {skip} }
    }
```

The probability of deleting a tree with root \( x \) is at least \( (1 - p)^{\text{size of the heap}} \).

\[
\text{wlp} \ [\text{delete}(x)] (\text{emp}) \geq \ [\text{tree}(x)] \cdot (1 - p)^{\text{size}}
\]
Example III: Randomized Meldable Heaps

A more formal proof of a less simple randomized data structure (courtesy of Hannah Arndt)

```plaintext
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)
    }
}
```

The expected length of a path from the root to a leaf is logarithmic in the number of nodes of the input tree.
Conclusion

Quantitative Separation Logic... 

as an assertion language
as a verification system
as a conservative, sound extension of separation logic
applied to example programs

Future Work

Automation (quantitative symbolic heaps? quantitative entailments?)
Concurrency (previous talk?)

Thank you for listening!
Conclusion

Quantitative Separation Logic... as an assertion language
as a verification system
as a conservative, sound extension of separation logic
applied to example programs

Future Work

Automation (quantitative symbolic heaps? quantitative entailments?)
Concurrency (previous talk?)
Conclusion

Quantitative Separation Logic. . .

- as an assertion language
- as a verification system
- as a conservative, sound extension of separation logic
- applied to example programs

Future Work

- Automation (quantitative symbolic heaps? quantitative entailments?)
- Concurrency (previous talk?)

Thank you for listening!
For **bounded** expectations $f$, the quantitative separating implication is

$$ f \rightarrow g = \lambda(s, h). \inf \left\{ \frac{g(s, h \star h')}{f(s, h')} \mid \text{dom}(h) \cap \text{dom}(h') = \emptyset \text{ and } f(s, h') > 0 \right\} $$

For **predicates** $[\varphi]$, the quantitative separating implication is

$$ [\varphi] \rightarrow g = \lambda(s, h). \inf \{ g(s, h \star h') \mid \text{dom}(h) \cap \text{dom}(h') = \emptyset \text{ and } [\varphi](s, h') = 1 \} $$
Example IV: Randomized List Extension

\[
\text{extendList}(x)\{ \\
\quad c := 1 \triangleright \\
\quad \text{while}(c = 1) \{ \\
\quad \quad \{ c := 0 \} [0.5] \{ c := 1 \triangleright x := \text{new}(x) \} \\
\quad \} \\
\}
\]

This program terminates \textbf{almost-surely}, but \textbf{not certainly}.

In expectation, the length of list \(x\) is increased by at most one.

\[
\text{wp} [\text{extendList}(x)] (\text{len}(x, 0)) \leq \text{len}(x, 0) + 1
\]
Example V: Lossy List Reversal

\[
\text{lossyReversal}(hd) \{
    r := 0, \\
\text{while}(hd \neq 0) \{
    t := *hd, \\
    \{ \begin{align*}
    *hd &:= r, \\
    r &:= hd
    \end{align*} \} [0.5] \{ \text{free}(hd) \} \\
    hd := t
  \}
\}
\]

In expectation, the length of reversed list \( r \) is at most half the length of the original list.

\[
\text{wp} \left[ \text{lossyReversal}(hd) \right] (\text{len}(r, 0)) \leq 0.5 \cdot [hd \neq 0] \cdot \text{len}(hd, 0)
\]