Quantitative Separation Logic
A Logic for Reasoning about Probabilistic Pointer Programs

Kevin Batz  Benjamin Lucien Kaminski
Joost-Pieter Katoen  Christoph Matheja\textsuperscript{1}  Thomas Noll

ROCKS Meeting

October 18, 2018, Freising, Germany

\textsuperscript{1}Supported by DFG Grant NO 401/2-1
Goal: Formal Program Verification

- on source code level
- without ad-hoc arguments
- compositional
Goal: Formal Program Verification

on source code level
without ad-hoc arguments
compositional

Hoare logic
Goal: Formal Program Verification

on source code level  without ad-hoc arguments  compositional

Hoare logic

The world has pointers in it
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 for pointer programs
Goal: Formal Program Verification

on source code level without ad-hoc arguments compositional

Hoare logic

The world has pointers in it

separation logic
Goal: Formal Program Verification

on source code level without ad-hoc arguments compositional

The world has pointers in it

separation logic

Hoare logic

The world is quantitative
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.”
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:**

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

Weakest preexpectations\(^1\) enable *compositional* and *quantitative* reasoning for *probabilistic* programs

\(^1\) and Kozen’s PPDL
Goal: Formal Program Verification

- on source code level
- without ad-hoc arguments
- compositional

The world has pointers in it

Hoare logic

separation logic

preexpectations

The world is quantitative
Goal: Formal Program Verification

- on source code level
- without ad-hoc arguments
- compositional

- Hoare logic
  - The world has pointers in it
  - separation logic
  - The world is quantitative

- The world is quantitative

- preexpectations
  - The world is quantitative
Goal: Formal Program Verification

on source code level  without ad-hoc arguments  compositional

The world has pointers in it

Hoare logic

The world is quantitative

separation logic

The world is quantitative

preexpectations

The world has pointers in it

The world has pointers in it

The world is quantitative
Goal: Formal Program Verification

- on source code level
- without ad-hoc arguments
- compositional

Hoare logic

separation logic

- The world has pointers in it
- The world is quantitative

preexpectations

- The world is quantitative
- The world has pointers in it

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}@csail.mit.edu

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 a developer to perform sound and verified 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 Reasoning about Programs

General Terms Languages, Reliability, Verification

Keywords Approximate Computing

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 [10] 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 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 execute 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 that 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 computations [1, 2, 18, 29, 42–44, 59] and checkable computations [8, 9, 35, 55]. Their results show that it is possible to exploit these properties for a variety of purposes — increased performance, reduced energy consumption, increased adaptability, and increased fault tolerance. One key aspect of such computations is that they typically contain critical regions (which must execute without error) and approximate regions (which can execute acceptably even in the presence of occasional errors) [16, 59]. To support such computations, researchers have proposed energy-efficient architectures that, because they omit some...
Goal: Formal Program Verification

Verifying Quantitative Reliable That Execute on Unreliable

Michael Carbin Sasa Misailovic
MIT CSAIL
{
{carbin, misailovic, misailovic}

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 ranking 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 verified 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 the 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 Reasoning about Programs

General Terms
Languages, Reliability, Verification

Keywords
Approximate Computing

1. Introduction
Emerging high-performance systems are emerging as a result of the anticipated increase in the frequency of soft errors in hardware [23] and software [20, 53, 57, 64]. These include emerging architectures and system impacts, which are becoming major goals when designing new devices. System reliability is a major challenge in the design of high-performance architectures.

Many computations, however, can tolerate occasional hardware [23] and software [20, 53, 57, 64]. These techniques can be used to approximately tolerate soft errors. For example, programmers can add checks to detect runtime soft errors. However, this approach is not effective when the occurrence of soft errors is not predictable. In such cases, the programmer must manually estimate the probability that a computation will fail.

Binary trees can be used for representing abstract data structures such as dictionaries and ordered sets. Each work well when the elements are inserted in a random order. Some sequences of operations, such as inserting the elements in a tree, produce programs that perform very poorly. If it were possible to randomly generate the list of items to be inserted, the trees would work well with high probability for any input sequence. However, since an arbitrary tree is impractical, balanced tree algorithms rearrange 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 do not maintain balance, but they have the same asymptotic time complexity as balanced trees. A node's ith level of a node, chosen randomly when the node is inserted, is the number of pointers. The number of pointers is chosen randomly from the set of integers from 1 to k, where k is the number of levels. The simplicity of skip list data structures makes them well suited for real-world applications. Skip lists are particularly useful when the elements are inserted in a random order.

William Pugh

Binary trees can be used for representing abstract data structures such as dictionaries and ordered sets. Each work well when the elements are inserted in a random order. Some sequences of operations, such as inserting the elements in a tree, produce programs that perform very poorly. If it were possible to randomly generate the list of items to be inserted, the trees would work well with high probability for any input sequence. However, since an arbitrary tree is impractical, balanced tree algorithms rearrange 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 do not maintain balance, but they have the same asymptotic time complexity as balanced trees. A node's ith level of a node, chosen randomly when the node is inserted, is the number of pointers. The number of pointers is chosen randomly from the set of integers from 1 to k, where k is the number of levels. The simplicity of skip list data structures makes them well suited for real-world applications. Skip lists are particularly useful when the elements are inserted in a random order.

William Pugh

Binary trees can be used for representing abstract data structures such as dictionaries and ordered sets. Each work well when the elements are inserted in a random order. Some sequences of operations, such as inserting the elements in a tree, produce programs that perform very poorly. If it were possible to randomly generate the list of items to be inserted, the trees would work well with high probability for any input sequence. However, since an arbitrary tree is impractical, balanced tree algorithms rearrange 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 do not maintain balance, but they have the same asymptotic time complexity as balanced trees. A node's ith level of a node, chosen randomly when the node is inserted, is the number of pointers. The number of pointers is chosen randomly from the set of integers from 1 to k, where k is the number of levels. The simplicity of skip list data structures makes them well suited for real-world applications. Skip lists are particularly useful when the elements are inserted in a random order.

William Pugh

Binary trees can be used for representing abstract data structures such as dictionaries and ordered sets. Each work well when the elements are inserted in a random order. Some sequences of operations, such as inserting the elements in a tree, produce programs that perform very poorly. If it were possible to randomly generate the list of items to be inserted, the trees would work well with high probability for any input sequence. However, since an arbitrary tree is impractical, balanced tree algorithms rearrange 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 do not maintain balance, but they have the same asymptotic time complexity as balanced trees. A node's ith level of a node, chosen randomly when the node is inserted, is the number of pointers. The number of pointers is chosen randomly from the set of integers from 1 to k, where k is the number of levels. The simplicity of skip list data structures makes them well suited for real-world applications. Skip lists are particularly useful when the elements are inserted in a random order.

William Pugh

Binary trees can be used for representing abstract data structures such as dictionaries and ordered sets. Each work well when the elements are inserted in a random order. Some sequences of operations, such as inserting the elements in a tree, produce programs that perform very poorly. If it were possible to randomly generate the list of items to be inserted, the trees would work well with high probability for any input sequence. However, since an arbitrary tree is impractical, balanced tree algorithms rearrange 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 do not maintain balance, but they have the same asymptotic time complexity as balanced trees. A node's ith level of a node, chosen randomly when the node is inserted, is the number of pointers. The number of pointers is chosen randomly from the set of integers from 1 to k, where k is the number of levels. The simplicity of skip list data structures makes them well suited for real-world applications. Skip lists are particularly useful when the elements are inserted in a random order.

William Pugh

Binary trees can be used for representing abstract data structures such as dictionaries and ordered sets. Each work well when the elements are inserted in a random order. Some sequences of operations, such as inserting the elements in a tree, produce programs that perform very poorly. If it were possible to randomly generate the list of items to be inserted, the trees would work well with high probability for any input sequence. However, since an arbitrary tree is impractical, balanced tree algorithms rearrange 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 do not maintain balance, but they have the same asymptotic time complexity as balanced trees. A node's ith level of a node, chosen randomly when the node is inserted, is the number of pointers. The number of pointers is chosen randomly from the set of integers from 1 to k, where k is the number of levels. The simplicity of skip list data structures makes them well suited for real-world applications. Skip lists are particularly useful when the elements are inserted in a random order.

William Pugh

Binary trees can be used for representing abstract data structures such as dictionaries and ordered sets. Each work well when the elements are inserted in a random order. Some sequences of operations, such as inserting the elements in a tree, produce programs that perform very poorly. If it were possible to randomly generate the list of items to be inserted, the trees would work well with high probability for any input sequence. However, since an arbitrary tree is impractical, balanced tree algorithms rearrange 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 do not maintain balance, but they have the same asymptotic time complexity as balanced trees. A node's ith level of a node, chosen randomly when the node is inserted, is the number of pointers. The number of pointers is chosen randomly from the set of integers from 1 to k, where k is the number of levels. The simplicity of skip list data structures makes them well suited for real-world applications. Skip lists are particularly useful when the elements are inserted in a random order.
Goal: Formal Program Verification

### Verifying Quantitative Reliability That Execute on Unreliable Hardware

Michael Carbin  
Sasa Misailovic  
MIT CSAIL  
\{carbin, misail, rinard\}@csail.mit.edu

#### 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 analysis) 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 verified 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 meets its reliability specification when executed on the underlying unreliable hardware platform. We demonstrate the applicability of quantitative reliability analysis on six computations implemented in Rely.

#### Categories and Subject Descriptors

F.1.1 [Languages and Lanam Systems]: Language Constructs and Features  
E.1.1 (Performance of Systems): Reliability  
D.4.0 [Software Engineering]: Program Verification

#### General Terms

Languages, Reliability, Verification

#### Keywords

Approximate Computing

---

#### 1. Introduction

In this paper we present a randomized approach to the problem of efficient probabilistic priority queue implementation. The operations supported by this data structure are the following: (i)

1. **MaxQ(\(Q\))** returns the minimum item from priority queue \(Q\)
2. **DeleteMax(\(Q\))** deletes and returns the minimum item from priority queue \(Q\)
3. **Insert(\(Q\); \(e\))** inserts item \(e\) into priority queue \(Q\)
4. **Delete(\(Q\); \(e\))** replaces item \(e\) by \(e\) in priority queue \(Q\) provided \(e\) and the location of \(e\) in \(Q\) is known.
5. **Delete(\(Q\); \(e\))** deletes item \(e\) from priority queue \(Q\) provided the location of \(e\) in \(Q\) is known.

We assume the following properties that are satisfied by all implementations:

1. The list operations are never concurrently executed.
2. In existing priority queue implementations the operations are 
   * **MaxQ(\(Q\))** returns the minimum item from priority queue \(Q\)
   * **DeleteMax(\(Q\))** deletes and returns the minimum item from priority queue \(Q\)
   * **Insert(\(Q\); \(e\))** inserts item \(e\) into priority queue \(Q\)
   * **Delete(\(Q\); \(e\))** replaces item \(e\) by \(e\) in priority queue \(Q\) provided \(e\) and the location of \(e\) in \(Q\) is known.

#### 2. Randomized Meldable Priority Queues

**A. Gambs and A. Misailovic**

In existing priority queue implementations the operations are:

1. **MaxQ(\(Q\))** returns the minimum item from priority queue \(Q\)
2. **DeleteMax(\(Q\))** deletes and returns the minimum item from priority queue \(Q\)
3. **Insert(\(Q\); \(e\))** inserts item \(e\) into priority queue \(Q\)
4. **Delete(\(Q\); \(e\))** replaces item \(e\) by \(e\) in priority queue \(Q\) provided \(e\) and the location of \(e\) in \(Q\) is known.

#### 3. Example for Skip Lists

**William Pugh**

Binary trees can be used for representing abstract data types such as dictionaries and ordered lists. They work on source code level and can be used for representing data in an ordered list. Binary trees can be used for representing abstract data types such as dictionaries and ordered lists. They work on source code level and can be used for representing data in an ordered list. Binary trees can be used for representing abstract data types such as dictionaries and ordered lists. They work on source code level and can be used for representing data in an ordered list.

In existing priority queue implementations the operations are:

1. **MaxQ(\(Q\))** returns the minimum item from priority queue \(Q\)
2. **DeleteMax(\(Q\))** deletes and returns the minimum item from priority queue \(Q\)
3. **Insert(\(Q\); \(e\))** inserts item \(e\) into priority queue \(Q\)
4. **Delete(\(Q\); \(e\))** replaces item \(e\) by \(e\) in priority queue \(Q\) provided \(e\) and the location of \(e\) in \(Q\) is known.

#### 4. Example for Skip Lists

**William Pugh**

Binary trees can be used for representing abstract data types such as dictionaries and ordered lists. They work on source code level and can be used for representing data in an ordered list. Binary trees can be used for representing abstract data types such as dictionaries and ordered lists. They work on source code level and can be used for representing data in an ordered list. Binary trees can be used for representing abstract data types such as dictionaries and ordered lists. They work on source code level and can be used for representing data in an ordered list.
Quantitative Separation Logic (QSL)

Assertion language

Verification system

Theorems
QSL as an Assertion Language
States and Expectations
States and Expectations

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

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

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

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

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

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

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

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

Examples:
States and 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:** \( f : States \to \mathbb{R}_{\geq 0}^{\infty} \)

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

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

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

Examples:
\[
x = \lambda(s, h). \ s(x) \\
\text{size} = \lambda(s, h). \ |\text{dom}(h)|
\]
States and Expectations

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

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

\textbf{Examples:}

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

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

\textit{Atomic separation logic formulas are expectations}
States and Expectations

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

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

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

Atomic separation logic formulas are expectations
\[\text{emp} = \lambda(s, h). \begin{cases} 1 & \text{if } \text{dom}(h) = \emptyset \\ 0 & \text{otherwise} \end{cases}\]
States and Expectations

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

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

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

Atomic separation logic formulas are expectations

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

\[[x \mapsto y] = \lambda(s, h). \begin{cases} 1 & \text{if } \text{dom}(h) = \{s(x)\}, h(s(x)) = s(y) \\ 0 & \text{otherwise} \end{cases}\]
What about Connectives?

Classical conjunction:

\[ \varphi \land \psi \]
What about Connectives?

Classical conjunction:

\( \varphi \land \psi \)

Quantitative conjunction:

\[
f \cdot g = \lambda(s, h). f(s, h) \cdot g(s, h)
\]
What about Connectives?

Classical conjunction:

\[ \varphi \land \psi \]

Quantitative conjunction:

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

Remark:

\[ [\varphi \land \psi] = [\varphi] \cdot [\psi] \]
Separating Conjunction

Classical separating conjunction:

\[(s, h) \models \varphi \star \psi \text{ iff } \exists h_1, h_2 : h = h_1 \star h_2 \text{ and } (s, h_1) \models \varphi \text{ and } (s, h_2) \models \psi\]
Classical separating conjunction:

\[(s, h) \models \varphi \ast \psi \quad \text{iff} \quad \exists h_1, h_2 : h = h_1 \ast h_2 \quad \text{and} \quad (s, h_1) \models \varphi \quad \text{and} \quad (s, h_2) \models \psi\]
Separating Conjunction

Classical separating conjunction:

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

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

⋆ and —⋆ are adjoint:

\[(f \star g) \leq r \text{ iff } f \leq (g \rightarrow⋆ r)\]
Separating Implication

⋆ and —⋆ are adjoint:

\[(f \star g) \leq r \iff f \leq (g \rightarrow* r)\]

Intuition: \[x - y \leq z \iff x \leq y + z\]
Separating Implication

⋆ and ⊢⋆ are adjoint:

\[(f \star g) \leq r \iff f \leq (g \rightarrow \star r)\]

Intuition: \(x - y \leq z \iff x \leq y + z\)

Classical separating implication:

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

⋆ and →⋆ are adjoint:

\[(f \star g) \leq r \iff f \leq (g \to ⋆ r)\]

Intuition: \[x \to y \leq z \iff x \leq y + z\]

Classical separating implication:

\[(s, h) \models \varphi \to ⋆ \psi \iff \forall h' \text{ with } h \perp h' \text{ and } (s, h') \models \varphi: (s, h \star h') \models \psi\]

Quantitative separating implication:

\[f \to ⋆ g = \lambda(s, h). \inf \left\{ \frac{g(s, h \star h')}{f(s, h')} \mid h \perp h' \text{ and } f(s, h') > 0 \right\}\]
Separating Implication

⋆ and →⋆ are adjoint:

\[(f \ast g) \leq r \iff f \leq (g \rightarrow⋆ r)\]

Intuition: \(x - y \leq z \iff x \leq y + z\)

Classical separating implication:

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

Quantitative separating implication:

\[f \rightarrow⋆ g = \lambda(s, h) \cdot \inf \left\{ \frac{g(s, h \ast h')}{f(s, h')} \mid h \perp h' \text{ and } f(s, h') > 0 \right\}\]

\[[\varphi] \rightarrow⋆ g = \lambda(s, h) \cdot \inf \{ g(s, h \ast h') \mid h \perp h' \text{ and } [\varphi](s, h') = 1 \}\]
QSL as a Verification System
The Weakest Preexpectation Transformer

\[ \text{wp} [c] : \text{Expectations} \rightarrow \text{Expectations} \]
The Weakest Preexpectation Transformer

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

\[ c \quad f \]

\( f \) is a \textit{postexpectation} evaluated in \textit{final states}. 
The Weakest Preexpectation Transformer

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

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

\[ f \text{ is a postexpectation evaluated in final states.} \]
The Weakest Preexpectation Transformer

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

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

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

\[ \text{f is a postexpectation evaluated in final states.} \]

14 Christoph Matheja | Quantitative Separation Logic | ROCKS 2018
Weakest Preexpectation of Probabilistic Choice

\[ \{ c_1 \} \; [p] \; \{ c_2 \} \]
Weakest Preexpectation of Probabilistic Choice

\[ \langle \{ c_1 \} \| p \| \{ c_2 \}, s, h \rangle \]

\[ p \rightarrow \langle c_1, s, h \rangle \]

\[ 1 - p \rightarrow \langle c_2, s, h \rangle \]
Weakest Preexpectation of Probabilistic Choice

\[ \wp \left[ \left\{ c_1 \right\} \left[ p \right] \left\{ c_2 \right\} \right] (f) = p \cdot \wp \left[ c_1 \right] (f) + (1 - p) \cdot \wp \left[ c_2 \right] (f) \]
Weakest Preexpectation of Memory Allocation

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

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

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

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

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

\[
\begin{array}{c|c|c}
  x: & y: & \cdots \\
  v & w & \quad a & b & c & \cdots \\
  \end{array}
\]

\[
\begin{array}{c|c|c}
  \downarrow s: & \downarrow h: \\
  \end{array}
\]

\[
\begin{array}{c|c|c}
  \downarrow s_\perp: & \downarrow h_\perp: \\
  \end{array}
\]
Weakest Preexpectation of Memory Allocation

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

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

\[ h : \]
\[ x : y : \ldots \]
\[ \alpha : \beta : \gamma : \ldots \]

\[ s_\downarrow : \]
\[ x : y : \ldots \]
\[ v \quad w \quad \ldots \]
\[ \alpha : \beta : \gamma : \ldots \]

\[ h_\downarrow : \]
\[ x : y : \ldots \]
\[ v \quad w \quad \ldots \]
\[ \alpha : \beta : \gamma : \ldots \]

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

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

\[ s: \quad h: \]

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

\[ s_\perp: \quad h_\perp: \]

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

\[ v \quad \ast \quad \] $s(e)$

\[ \nu = 1 \]

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

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

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

<table>
<thead>
<tr>
<th>s:</th>
<th>h:</th>
</tr>
</thead>
<tbody>
<tr>
<td>x:</td>
<td>y:</td>
</tr>
<tr>
<td>u</td>
<td>w</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>s_:</th>
<th>h_:</th>
</tr>
</thead>
<tbody>
<tr>
<td>x:</td>
<td>y:</td>
</tr>
<tr>
<td>v</td>
<td>w</td>
</tr>
</tbody>
</table>

\[ s \downarrow: h \downarrow: \]

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

\[ v = 1 \]

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

\[ v = 2 \]

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

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

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

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

\[ s: \quad h: \]

\[ s_\perp: \quad h_\perp: \]

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

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

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

\[ \vdots \]

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

16 Christoph Matheja | Quantitative Separation Logic | ROCKS 2018
Weakest Preexpectation of Memory Allocation

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

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

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

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

\[ \vdots \]
Weakest Preexpectation of Memory Allocation

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

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

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

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

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

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

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

\[ \vdots \]
Weakest Preexpectation of Memory Allocation

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

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

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

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

\[ \vdots \]
Weakest Preexpectation of Memory Allocation

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

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

\[ \text{inf}_{v \in \text{loc}}: [v \mapsto e] \rightarrow f[x/v] \]

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

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

valid extension

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

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

\[ v = 1 \]

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

\[ v = 2 \]

\[ \vdots \]
The Weakest Preexpectation Calculus

<table>
<thead>
<tr>
<th>Syntax</th>
<th>wp $<a href="f">c</a>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>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>$\text{wp } [c_1] (\text{wp } <a href="f">c_2</a>)$</td>
</tr>
<tr>
<td>if ($\xi$) {$c_1$} else {$c_2$}</td>
<td>$[\xi] \cdot \text{wp } <a href="f">c_1</a> + [\neg \xi] \cdot \text{wp } <a href="f">c_2</a>$</td>
</tr>
<tr>
<td>while ($\xi$) {$c'$}</td>
<td>$\text{lfp } g. \ [\neg \xi] \cdot f + [\xi] \cdot \text{wp } <a href="g">c'</a>$</td>
</tr>
<tr>
<td>{$c_1$} [p] {$c_2$}</td>
<td>$p \cdot \text{wp } <a href="f">c_1</a> + (1 - p) \cdot \text{wp } <a href="f">c_2</a>$</td>
</tr>
<tr>
<td>$x := \text{new}(e)$</td>
<td>$\inf_{v \in \text{loc}(e)} [v \mapsto e] \star f[x/v]$</td>
</tr>
<tr>
<td>$x := *e$</td>
<td>$\sup_{v \in \mathbb{Z}} [e \mapsto v] \star ([e \mapsto v] \star f[x/v])$</td>
</tr>
<tr>
<td>$*e := e'$</td>
<td>$[e \mapsto -] \star ([e \mapsto e'] \star f)$</td>
</tr>
<tr>
<td>free$(e)$</td>
<td>$[e \mapsto -] \star f$</td>
</tr>
</tbody>
</table>
Theorems
QSL conservatively extends separation logic

\[ \varphi, \psi \text{ SL formulas} \]
QSL conservatively extends separation logic

$\varphi, \psi$ SL formulas

$[\varphi], [\psi]$ corresponding QSL formulas
QSL conservatively extends separation logic

\( \varphi, \psi \) SL formulas

\([\varphi], [\psi]\) corresponding QSL formulas

\((s, h)\) a state
QSL conservatively extends separation logic

$\phi, \psi$ SL formulas

$([\phi], [\psi])$ corresponding QSL formulas

$(s, h)$ a state

$c$ a non-prob. pointer program
QSL conservatively extends separation logic

\( \varphi, \psi \) SL formulas

\( (s, h) \) a state

\( [\varphi], [\psi] \) corresponding QSL formulas

\( c \) a non-prob. pointer program

1) \( [\varphi](s, h) \in \{0, 1\} \)
QSL conservatively extends separation logic

\[ \varphi, \psi \text{ SL formulas} \quad \begin{array}{c} \square \varphi, \square \psi \text{ corresponding QSL formulas} \\ (s, h) \text{ a state} \quad c \text{ a non-prob. pointer program} \end{array} \]

1) \[ \square \varphi(s, h) \in \{0, 1\} \]

2) \[ (s, h) \models \varphi \text{ iff } \square \varphi(s, h) = 1 \]
QSL conservatively extends separation logic

\( \varphi, \psi \) SL formulas

\( [\varphi], [\psi] \) corresponding QSL formulas

(\( s, h \)) a state

c a non-prob. pointer program

1) \( [\varphi](s, h) \in \{0, 1\} \)

2) \( (s, h) \models \varphi \iff [\varphi](s, h) = 1 \)

3) \( \{ \varphi \} c \{ \psi \} \) valid \( \iff [\varphi] \leq \text{wp} \left[ c \right] ([\psi]) \)
QSL’s weakest preexpectations are sound

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

$$\text{wp } \llbracket c \rrbracket (f)(s, h) = \frac{\text{ExpRew}[f](c, s, h)}{\text{of operational MDP}}.$$
QSL’s weakest preexpectations are sound

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

$$wp \left[ c \right] (f) (s, h) = \underbrace{\text{ExpRew} [f] (c, s, h)}_{\text{of operational MDP}}.$$
QSL’s weakest preexpectations are sound

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

$$\text{wp} \left[ c \right] (f) (s, h) = \frac{\text{ExpRew}[f](c, s, h)}{\text{of operational MDP}}.$$
The Quantitative Frame Rule

The classical frame rule

\[\{ \varphi \} \quad c \quad \{ \psi \} \quad \frac{}{\{ \varphi \star \rho \} \quad c \quad \{ \psi \star \rho \}}\]

Mod \( (c) \cap \text{Vars} (\rho) = \emptyset \)

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

\[\text{wp} (J \quad c \quad K) (f \star g) \leq \text{wp} (J \quad c \quad K) (f \star g)\]

The converse direction (\( \geq \)) breaks already in the qualitative case!
The Quantitative Frame Rule

The classical frame rule

\[
\begin{array}{c}
\{ \varphi \} \ c \ \{ \psi \} \\
\{ \varphi \star \rho \} \ c \ \{ \psi \star \rho \}
\end{array}
\]

\[\text{Mod}(c) \cap \text{Vars}(\rho) = \emptyset\]
The Quantitative Frame Rule

The classical frame rule

\[
\frac{\{ \varphi \} \ c \ \{ \psi \} \quad Mod(c) \cap \text{Vars}(\rho) = \emptyset}{\{ \varphi \ast \rho \} \ c \ \{ \psi \ast \rho \}}
\]

The quantitative frame rule

For all expectations \( f, g \) with \( Mod(c) \cap \text{Vars}(g) = \emptyset \),
The Quantitative Frame Rule

The classical frame rule

\[
\begin{array}{c}
\{ \varphi \} \ c \ \{ \psi \} \\
\{ \varphi \star \rho \} \ c \ \{ \psi \star \rho \}
\end{array}
\]
\[
\text{Mod}(c) \cap \text{Vars}(\rho) = \emptyset
\]

The quantitative frame rule

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

\[
\text{wp} \ [c] \ (f) \star g \leq \text{wp} \ [c] \ (f \star g).
\]
The Quantitative Frame Rule

The classical frame rule

\[
\frac{\{ \varphi \} \ c \ \{ \psi \}}{\{ \varphi \star \rho \} \ c \ \{ \psi \star \rho \}} \quad \text{Mod}(c) \cap \text{Vars}(\rho) = \emptyset
\]

The quantitative frame rule

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

\[
\text{wp} \ \llbracket c \rrbracket \ (f) \star g \leq \text{wp} \ \llbracket c \rrbracket \ (f \star g).
\]

The converse direction (\( \geq \)) breaks already in the \textbf{qualitative} case!
Case Studies
Array Randomization à la Cormen et al.

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

wp [call randomize(array, n)] ([array ↦ α₀, ..., α_{n-1}]) ≤ \frac{1}{n!}
```
procedure delete(x) { \( t(x) \)
  if (\( x \neq 0 \)) {
    \{ skip \} [p] {
      \{ skip \} [p] {
        \( l := \ast x \) ; \( r := \ast (x + 1) \) ; [x \mapsto -,-] \ast [t(l) \ast t(r)]
        call delete(l) ; [x \mapsto -,-] \ast t(r)
        call delete(r) ; [x \mapsto -,-] \ast [\text{emp}]
        free(x) ; free(x + 1) \} [\text{emp}]
      } else {skip} } [\text{emp}]
  } [\text{emp}]

\[ \text{wp} [\text{call delete}(x)] (\text{[emp]}) \geq [\text{tree}(x)] \cdot (1 - p)^{\text{size}} = t(x) \]
Randomized Meldable Heaps (in progress, courtesy Hannah Arndt)

procedure randomLeaf (root)
{
    nextL := *root ;
    nextR := *(root + 1) ;
    if (nextL = 0 and nextR = 0) {
    return root
    } else {
        { next := nextL } [0.5] { next := nextR } ;
        return call randomLeaf (root)
    }
}

wp [[leaf = call randomLeaf (root)]] ([tree](root) · pathlen(tree, leaf)) ≤ log(#nodes(root) + 1)
Conclusion

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

Quantitative Separation Logic . . .

as an assertion language

as a verification system

as a conservative, sound extension of separation logic

Further Reading

Conclusion

Quantitative Separation Logic . . .

as an assertion language

as a verification system

as a conservative, sound extension of separation logic

Further Reading


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!