Quantitative Separation Logic

A Logic for Reasoning about Probabilistic Pointer Programs

KEVIN BATZ, RWTH Aachen University, Germany
BENJAMIN LUCIEN KAMINSKI, RWTH Aachen University, Germany
JOOST-PIETER KATOEN, RWTH Aachen University, Germany
CHRISTOPH MATHEJA, RWTH Aachen University, Germany
THOMAS NOLL, RWTH Aachen University, Germany

We present *quantitative separation logic* (QSL). In contrast to classical separation logic, QSL employs quantities which evaluate to real numbers instead of predicates which evaluate to Boolean values. The connectives of classical separation logic, separating conjunction and separating implication, are lifted from predicates to quantities. This extension is conservative: Both connectives are backward compatible to their classical analogs and obey the same laws, e.g. modus ponens, adjointness, etc.

Furthermore, we develop a weakest precondition calculus for quantitative reasoning about *probabilistic pointer programs* in QSL. This calculus is a conservative extension of both Ishtiaq's, O'Hearn's and Reynolds' separation logic for heap-manipulating programs and Kozen's / McIver and Morgan's weakest preexpectations for probabilistic programs. Soundness is proven with respect to an operational semantics based on Markov decision processes. Our calculus preserves O'Hearn's *frame rule*, which enables local reasoning. We demonstrate that our calculus enables reasoning about quantities such as the probability of terminating with an empty heap, the probability of reaching a certain array permutation, or the expected length of a list.

CCS Concepts: • **Theory of computation → Probabilistic computation**; **Logic and verification**; **Programming logic**; **Separation logic**; **Program semantics**; **Program reasoning**;

Additional Key Words and Phrases: quantitative separation logic, probabilistic programs, randomized algorithms, formal verification, quantitative reasoning

ACM Reference Format:
Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Thomas Noll. 2019. Quantitative Separation Logic: A Logic for Reasoning about Probabilistic Pointer Programs. *Proc. ACM Program. Lang.* 3, POPL, Article 34 (January 2019), 29 pages. https://doi.org/10.1145/3290347

1 INTRODUCTION

Randomization plays an important role in the construction of algorithms. It typically improves average-case performance at the cost of a worse best-case performance or at the cost of incorrect results occurring with low probability. The former is observed when, e.g., randomly picking the pivot in quicksort [Hoare 1962]. A prime example of the latter is Freivalds' matrix multiplication verification algorithm [Freivalds 1977].

Authors' addresses: Kevin Batz, RWTH Aachen University, Germany, kevin.batz@rwth-aachen.de; Benjamin Lucien Kaminski, RWTH Aachen University, Germany, benjamin.kaminski@cs.rwth-aachen.de; Joost-Pieter Katoen, RWTH Aachen University, Germany, katoen@cs.rwth-aachen.de; Christoph Matheja, RWTH Aachen University, Germany, matheja@cs.rwth-aachen.de; Thomas Noll, RWTH Aachen University, Germany, noll@cs.rwth-aachen.de.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2019 Copyright held by the owner/author(s).
2475-1421/2019/1-ART34
https://doi.org/10.1145/3290347

RIGHTSLINK

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

(a) Procedure to randomize an array of length n

```
procedure lossyReversal (hd) {
    r := 0 ;
    while ( hd ≠ 0 ) {
        t := <hd> ;
        { <hd> := r ;  } [1/2] { free(hd) }
        { r := hd      }
        hd := t
} }
```

(b) Lossy reversal of a list with head hd

Fig. 1. Examples of probabilistic programs. We write $<e>$ to access the value stored at address e.

Sophisticated algorithms often make use of *randomized data structures*. For instance, Pugh states that randomized skip lists enjoy "the same asymptotic expected time bounds as balanced trees and are faster and use less space" [Pugh 1990]. Other examples of randomized data structures include randomized splay trees [Albers and Karpinski 2002], treaps [Blelloch and Reid-Miller 1998] and randomized search trees [Aragon and Seidel 1989; Martínez and Roura 1998].

Randomized algorithms are conveniently described by probabilistic programs, i.e. programs with the ability to sample from a probability distribution, e.g. by flipping coins. While randomized algorithms have desirable properties, their verification often requires reasoning about programs that mutate dynamic data structures *and* behave probabilistically. Both tasks are challenging on their own and have been the subject of intensive research, see e.g. [Barthe et al. 2018; Chakarov and Sankaranarayanan 2013; Chatterjee et al. 2016; Kozen 1979; Krebbers et al. 2017; McIver et al. 2018; Ngo et al. 2018; O'Hearn 2012]. However, to the best of our knowledge, work on formal verification of programs that are *both* randomized *and* heap-manipulating is scarce. To highlight the need for *quantitative properties* and their formal verification in this setting let us consider three examples.

Example 1: Array randomization. A common approach to design randomized algorithms is to randomize the input and process it in a deterministic manner. For instance, the only randomization involved in algorithms solving the famous Secretary Problem (cf. [Cormen et al. 2009, Chapter 5.1]) is computing a random permutation of its input array. A textbook implementation (cf. [Cormen et al. 2009, Chapter 5.3]) of such a procedure `randomize` for an array of length n is depicted in Figure 1a. For each position in the array, the procedure uniformly samples a random number j in the remaining array between the current position i and the last position $n-1$. After that, the elements at position i and j are swapped. The procedure `randomize` is correct precisely if all outputs are equally likely. Thus, to verify correctness of this procedure, we inevitably have to reason about a probability, hence a *quantity*. In fact, each of the $n!$ possible permutations of the input array is computed by procedure `randomize` with probability at most $1/n!$.

Beyond randomized algorithms. Probabilistic programs are a powerful modeling tool that is not limited to randomized algorithms. Consider, for instance, approximate computing: Programs running on unreliable hardware, where instructions may occasionally return incorrect results, are naturally captured by probabilistic programs [Carbin et al. 2016]. Since incorrect results are unavoidable in such a scenario, the notion of a program's correctness becomes blurred: That is, quantifying (and minimizing) the probability of encountering a failure or the expected error of a program becomes crucial. The need for quantitative reasoning is also stressed by [Henzinger 2013]

RIGHTSLINK

who argues that "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 [...]."

Example 2: Faulty garbage collector. Consider a procedure $\texttt{delete}(x)$ that takes a tree with root x and recursively deletes all of its elements. This is a classical example due to [O'Hearn 2012; Reynolds 2002]. However, our procedure fails with some probability $p \in [0, 1]$ to continue deleting subtrees, i.e. running $\texttt{delete}(x)$ on a tree with root x does not necessarily result in the empty heap. If failures of $\texttt{delete}(x)$ are caused by unreliable hardware, they are unavoidable. Instead of proving a Boolean correctness property, we are thus interested in evaluating the reliability of the procedure by *quantifying the probability of collecting all garbage*. In fact, the probability of completely deleting a tree with root x containing n nodes is at least $(1 - p)^n$. Thus, to guarantee that a tree containing 100 elements is deleted at least with probability 0.90, the probability p must be below 0.00105305.

Example 3: Lossy list reversal. A prominent benchmark when analyzing heap-manipulating programs is in-place list-reversal (cf. [Atkey 2011; Krebbers et al. 2017; Magill et al. 2006]). Figure 1b depicts a *lossy* list reversal: The procedure `lossyReversal` traverses a list with head *hd* and attempts to move each element to the front of an initially empty list with head *r*. However, during each iteration, the current element is dropped with probability 1/2. This is modeled by a *probabilistic choice*, which either updates the value at address *hd* or disposes that address:

$$\{ < hd > \, := r \,;\, r \, := hd \} \; [\, ^1/_2 \,] \; \{ \texttt{free}(hd) \}$$

The procedure `lossyReversal` is not functionally correct in the sense that, upon termination, *r* is the head of the reversed initial list: Although the program never crashes due to a memory fault and indeed produces a singly-linked list, the length of this list varies between zero and the length of the initial list. A more sensible quantity of interest is the *expected*, i.e. average, *length of the reversed list*. In fact, the expected list length is *at most half* of the length of the original list.

Our approach. We develop a *quantitative separation logic* (QSL) for quantitative reasoning about heap-manipulating *and* probabilistic programs at source code level. Its distinguished features are:

- QSL *is quantitative*: It evaluates to a real number instead of a Boolean value. It is capable of specifying values of program variables, heap sizes, list lengths, etc.
- QSL *is probabilistic*: It enables reasoning about probabilistic programs, in particular about the *probability of terminating with a correct result*. It allows to express *expected values* of quantities, such as expected heap size or expected list length in a natural way.
- QSL *is a separation logic*: It conservatively extends separation logic (SL) [Ishtiaq and O'Hearn 2001; Reynolds 2002; Yang and O'Hearn 2002]. Our quantitative analogs of SL's key operators, i.e. separating conjunction $\star$ and separating implication $\mathrel{-\!\!\star}$, preserve virtually all properties of their Boolean versions.

For program verification, separation logic is often used in a (forward) Floyd-Hoare style. For probabilistic programs, however, backward reasoning is more common. In fact, certain forward-directed predicate transformers do not exist when reasoning about probabilistic programs [Jones 1990, p. 135]. We develop a (backward) weakest-precondition style calculus that uses QSL to verify probabilistic heap-manipulating programs. This calculus is a marriage of the weakest preexpectation calculus by [McIver and Morgan 2005] and separation logic à la [Ishtiaq and O'Hearn 2001; Reynolds 2002]. In particular:

- Our calculus is a *conservative extension of two approaches*: For programs that never access the heap, we obtain the calculus of McIver and Morgan. Conversely, for Boolean properties of ordinary programs, we recover exactly the wp-rules of Ishtiaq, O'Hearn, and Reynolds. QSL preserves virtually all properties of classical separation logic—including the *frame rule*.

RIGHTSLINK

- Our calculus is *sound* with respect to an operational semantics based on Markov decision processes. While this has been shown before for simple probabilistic languages (cf. [Gretz et al. 2014]), heap-manipulating statements introduce new technical challenges. In particular, allocating fresh memory yields *countably infinite nondeterminism*, which breaks continuity and rules out standard constructions for loops.
- We apply our calculus to analyze all aforementioned examples.

Outline. In Section 2, we present a probabilistic programming language with pointers together with an operational semantics. Section 3 introduces QSL as an assertion language. In Section 4, we develop a wp-style calculus for the quantitative verification of (probabilistic) programs with QSL. Furthermore, we prove soundness of our calculus and develop a *frame rule* for QSL. Section 5 discusses alternative design choices for wp-style calculi and Section 6 briefly addresses how recursive procedures are incorporated. In Section 7, we apply QSL to four case studies, including the three introductory examples. Finally, we discuss related work in Section 8 and conclude in Section 9.

Detailed proofs of all theorems are found in a separate technical report [Batz et al. 2018].

2 PROBABILISTIC POINTER PROGRAMS

We use a simple, imperative language à la Dijkstra's guarded command language with two distinguished features: First, we endow our programs with a probabilistic choice instruction. Second, we allow for statements that allocate, mutate, access, and dispose memory.

2.1 Syntax

The set of programs in *heap-manipulating probabilistic guarded command language*, denoted hpGCL, is given by the grammar

$c \longrightarrow$	$\mathtt{skip}$	(effectless program)	$\mid \{c\}\,[p]\,\{c\}$	(prob. choice)
	$\mid x := e$	(assignment)	$\mid x := \mathtt{new}(e_1, \ldots, e_n)$	(allocation)
	$\mid c\,;c$	(seq. composition)	$\mid \langle e \rangle := e'$	(mutation)
	$\mid \mathtt{if}\ (b)\ \{c\}\ \mathtt{else}\ \{c\}$	(conditional choice)	$\mid x := \langle e \rangle$	(lookup)
	$\mid \mathtt{while}\,(b)\,\{c\}$	(loop)	$\mid \mathtt{free}(e),$	(deallocation)

where x is a variable in the set Vars, $e, e', e_1, \ldots, e_n$ are arithmetic expressions, b is a predicate, i.e. an expression over variables evaluating to either true or false, and $p \in [0,\ 1] \cap \mathbb{Q}$ is a probability.

2.2 Program states

A *program state* $(s,\ h)$ consists of a *stack* s, i.e. a valuation of variables by integers, and a *heap* h modeling dynamically allocated memory. Formally, the set of *stacks* is given by $\mathcal{S} = \{\, s \mid s\colon \mathsf{Vars} \longrightarrow \mathbb{Z} \,\}$. Like in a standard RAM model, a heap consists of memory addresses that each store a value and is thus a *finite* mapping from addresses (i.e. natural numbers) to values (which may themselves be allocated addresses in the heap). Formally, the set of *heaps* is given by

$$\mathcal{H} \;=\; \{\, h \mid h\colon N \rightarrow \mathbb{Z},\ N \subseteq \mathbb{N}_{>0},\ |N| < \infty \,\}\,.$$

The 0 is excluded as a valid address in order to model e.g. null-pointer terminated lists. The set of *program states* is given by $\Sigma = \{\, (s, h) \mid s \in \mathcal{S},\ h \in \mathcal{H} \,\}$. Notice that expressions e and guards b may depend on variables only (i.e. they may *not* depend upon the heap) and thus their evaluation never causes any side effects. Side effects such as dereferencing unallocated memory can only occur *after* evaluating an expression and trying to access the memory at the evaluated address.

Given a program state (s, h), we denote by $s(e)$ the evaluation of expression e in s, i.e. the value that is obtained by evaluating e after replacing any occurrence of any variable x in e by the value

RIGHTSLINK

$s(x)$. By slight abuse of notation, we also denote the evaluation of a Boolean expression b by $s(b)$. Furthermore, we write $s\,[x/v]$ to indicate that we set variable x to value $v \in \mathbb{Z}$ in stack s, i.e.[1]

$$s\,[x/v] \;=\; \lambda\, y\,\textbf{.}\begin{cases} v, & \text{if } y = x \\ s(y), & \text{if } y \neq x. \end{cases}$$

For heap h, $h\,[u/v]$ is defined analogously. For a given heap $h\colon N \rightarrow \mathbb{Z}$, we denote by $\mathsf{dom}\,(h)$ its *domain* N. Furthermore, we write $\{u \mapsto v_1, \ldots, v_n\}$ as a shorthand for the heap h given by

$$\mathsf{dom}\,(h) \;=\; \{u, u+1, \ldots, u+n-1\}, \quad \forall k \in \{0, \ldots, n-1\}\colon h(u+k) \;=\; v_{k+1}.$$

Two heaps h_1, h_2 are *disjoint*, denoted $h_1 \perp h_2$, if their domains do not overlap, i.e. $\mathsf{dom}\,(h_1) \cap \mathsf{dom}\,(h_2) = \emptyset$. The *disjoint union* of two disjoint heaps $h_1\colon N_1 \rightarrow \mathbb{Z}$ and $h_2\colon N_2 \rightarrow \mathbb{Z}$ is given by

$$h_1 \star h_2\colon \mathsf{dom}\,(h_1) \,\dot{\cup}\, \mathsf{dom}\,(h_2) \rightarrow \mathbb{Z}, \qquad \big(h_1 \star h_2\big)(n) \;=\; \begin{cases} h_1(n), & \text{if } n \in \mathsf{dom}\,(h_1) \\ h_2(n), & \text{if } n \in \mathsf{dom}\,(h_2)\,. \end{cases}$$

We denote by $h_\emptyset$ the *empty heap* with $\mathsf{dom}\,(h_\emptyset) = \emptyset$. Note that $h \star h_\emptyset = h_\emptyset \star h = h$ for any heap h. We define *heap inclusion* as $h_1 \subseteq h_2$ iff $\exists\, h_1' \perp h_1\colon h_1 \star h_1' = h_2$. Finally, we use the *Iverson bracket* [Knuth 1992] notation $[\varphi]$ to associate with predicate φ its indicator function. Formally,

$$[\varphi]: \quad \Sigma \rightarrow \{0,\ 1\}, \quad [\varphi]\,(s,h) \;=\; \begin{cases} 1, & \text{if } (s,\ h) \models \varphi \\ 0, & \text{if } (s,\ h) \not\models \varphi, \end{cases}$$

where $(s,h) \models \varphi$ denotes that φ evaluates to true in (s,h). Notice that while predicates may generally speak about stack-heap pairs, guards in hpGCL-programs may only refer to the stack.

2.3 Semantics

We assign meaning to hpGCL-statements in terms of a small-step operational semantics, i.e. an execution relation $\rightarrow$ between *program configurations*, which consist of a program state and either a program that is still to be executed, a symbol $\Downarrow$ indicating successful termination, or a symbol $\lightning$ indicating a memory fault. Formally, the set of program configurations is given by

$$\mathsf{Conf} \;=\; (\mathsf{hpGCL} \cup \{\,\Downarrow,\, \lightning\,\}) \times \Sigma\,.$$

Since our programming language admits memory allocation and probabilistic choice, our semantics has to account for both *nondeterminism* (due to the fact that memory is allocated at nondeterministically chosen addresses) and *execution probabilities*. Our execution relation is hence of the form

$$\rightarrow\, \subseteq\; \mathsf{Conf} \times \mathbb{N} \times ([0,1] \cap \mathbb{Q}) \times \mathsf{Conf}\,,$$

where the second component is an *action* labeling the nondeterministic choice taken in the execution step and the third component is the execution step's probability.[2] We usually write $c, s, h \xrightarrow{n,p} c', s', h'$ instead of $((c,(s,h)), n, p, (c',(s',h'))) \in\, \rightarrow$. The operational semantics of hpGCL-programs, i.e. the execution relation $\rightarrow$, is determined by the rules in Figure 2. Let us briefly go over those rules. The rules for `skip`, assignments, conditionals, and loops are standard. In each case, the execution proceeds deterministically, hence all actions are labeled 0 and the execution probability is 1. For a probabilistic choice $\{\,c_1\,\}\ [\,p\,]\ \{\,c_2\,\}$ there are two possible executions: With probability p we execute c_1 and with probability $1-p$, we execute c_2.

[1] We use λ-expressions to denote functions: Function $\lambda X\textbf{.}\,f$ applied to an argument α evaluates to f in which every occurrence of X is replaced by α.

[2] For simplicity, we tacitly distinguish between the probabilities 0.5 and $1-0.5$ to deal with the corner case of two identical executions between the same configurations.

RIGHTSLINK

$$\frac{}{\mathtt{skip}, s, h \xrightarrow{0,1} \Downarrow, s, h} \qquad \frac{s(e) = v}{x := e, s, h \xrightarrow{0,1} \Downarrow, s\,[x/v]\,, h}$$

$$\frac{c_1, s, h \xrightarrow{a,p} \lightning, s, h}{c_1 \,;\, c_2, s, h \xrightarrow{a,p} \lightning, s, h} \qquad \frac{c_1, s, h \xrightarrow{a,p} \Downarrow, s', h'}{c_1 \,;\, c_2, s, h \xrightarrow{a,p} c_2, s', h'} \qquad \frac{c_1, s, h \xrightarrow{a,p} c_1', s', h'}{c_1 \,;\, c_2, s, h \xrightarrow{a,p} c_1' \,;\, c_2, s', h'}$$

$$\frac{s(b) = \mathsf{true}}{\mathtt{if}\,(\,b\,)\,\{\,c_1\,\}\;\mathtt{else}\;\{\,c_2\,\}, s, h \xrightarrow{0,1} c_1, s, h} \qquad \frac{s(b) = \mathsf{false}}{\mathtt{if}\,(\,b\,)\,\{\,c_1\,\}\;\mathtt{else}\;\{\,c_2\,\}, s, h \xrightarrow{0,1} c_2, s, h}$$

$$\frac{s(b) = \mathsf{false}}{\mathtt{while}\,(\,b\,)\,\{\,c\,\}, s, h \xrightarrow{0,1} \Downarrow, s, h} \qquad \frac{s(b) = \mathsf{true}}{\mathtt{while}\,(\,b\,)\,\{\,c\,\}, s, h \xrightarrow{0,1} c \,;\, \mathtt{while}\,(\,b\,)\,\{\,c\,\}, s, h}$$

$$\frac{}{\{\,c_1\,\}\,[\,p\,]\,\{\,c_2\,\}, s, h \xrightarrow{0,p} c_1, s, h} \qquad \frac{}{\{\,c_1\,\}\,[\,p\,]\,\{\,c_2\,\}, s, h \xrightarrow{0,1-p} c_2, s, h}$$

$$\frac{u, u+1, \ldots, u+n-1 \in \mathbb{N}_{>0} \setminus \mathsf{dom}\,(h) \qquad s(e_1) = v_1, \ldots, s(e_n) = v_n}{x := \mathsf{new}\,(e_1, \ldots, e_n), s, h \xrightarrow{u,1} \Downarrow, s\,[x/u]\,, h \star \{u \mapsto v_1, \ldots, v_n\}}$$

$$\frac{s(e) = u \in \mathsf{dom}\,(h) \qquad s(e') = v}{< e > := e', s, h \xrightarrow{0,1} \Downarrow, s, h\,[u/v]} \qquad \frac{s(e) \notin \mathsf{dom}\,(h)}{< e > := e', s, h \xrightarrow{0,1} \lightning, s, h}$$

$$\frac{s(e) = u \in \mathsf{dom}\,(h) \qquad h(u) = v}{x := < e >, s, h \xrightarrow{0,1} \Downarrow, s\,[x/v]\,, h} \qquad \frac{s(e) \notin \mathsf{dom}\,(h)}{x := < e >, s, h \xrightarrow{0,1} \lightning, s, h}$$

$$\frac{s(x) = u}{\mathtt{free}(x), s, h \star \{u \mapsto v\} \xrightarrow{0,1} \Downarrow, s, h} \qquad \frac{s(x) \notin \mathsf{dom}\,(h)}{\mathtt{free}(x), s, h \xrightarrow{0,1} \lightning, s, h}$$

Fig. 2. Inference rules determining the execution relation $\longrightarrow$.

The remaining statements access or manipulate memory. $x := \mathsf{new}\,(e_1, \ldots, e_n)$ allocates a block of n memory addresses and stores the first allocated address in variable x. Since allocated addresses are chosen nondeterministically by the memory allocator, there are countably infinitely many possible executions, which are each labeled by an action corresponding to the first allocated address. Under the assumption that an infinite amount of memory is available, *memory allocation cannot fail.* $< e > := e'$ attempts to write the value of e' to address e. If address e has not been allocated before, we encounter a memory fault, i.e. move to a configuration marked by $\lightning$. Conversely, $x := < e >$ assigns the value at address e to variable x. Again, failing to find address e on the heap leads to an error. Finally, $\mathtt{free}(e)$ disposes the memory cell at address e if it is present and fails otherwise.

Notice that no statement other than memory allocation introduces nondeterminism, i.e. entails an action label different from 0 Moreover, for every action $n \in \mathbb{N}$, we have

$$\sum_{c,s,h \xrightarrow{n,p} c',s',h'} p \in \{0, 1\},$$

where we set $\sum_{\emptyset} = 0$. Our execution relation thus describes a Markov Decision Process, which is an established model for probabilistic systems (cf. [Baier and Katoen 2008; Puterman 2005]).

RIGHTSLINK

3 QUANTITATIVE SEPARATION LOGIC

The term *separation logic* refers to both a logical assertion language as well as a Floyd-Hoare-style proof system for reasoning about pointer programs (cf. [Ishtiaq and O'Hearn 2001; Reynolds 2002]). In this section, we develop QSL in the sense of an assertion language. A proof system for reasoning about hpGCL programs is introduced in Section 4. The rationale of QSL is to combine concepts from two worlds:

(1) From separation logic (SL): *separating conjunction* ($\star$) and *separating implication* ($\mathrel{-\!\!\star}$).
(2) From probabilistic program verification: *expectations*.

Separating conjunction and implication are the two distinguished logical connectives featured in SL [Ishtiaq and O'Hearn 2001; Reynolds 2002]. Expectations [McIver and Morgan 2005] on the other hand take over the role of logical formulae when doing *quantitative reasoning* about probabilistic programs. In what follows, we gradually develop both a *quantitative separating conjunction* and a *quantitative separating implication* which each connect expectations instead of formulae (as in the classical setting).

3.1 Expectations

Floyd-Hoare logic [Hoare 1969] as well as Dijkstra's weakest preconditions [Dijkstra 1976] employ first-order logic for reasoning about the correctness of programs. For probabilistic programs, Kozen in his PPDL [Kozen 1983] was the first to generalize from predicates to measurable functions (or random variables). Later, [McIver and Morgan 2005] coined the term *expectation* for such functions. Here, we define the set $\mathbb{E}$ of expectations and the set $\mathbb{E}_{\leq 1}$ of one-bounded expectations as

$$\mathbb{E} \;=\; \left\{ X \;\middle|\; X\colon \Sigma \to \mathbb{R}_{\geq 0}^{\infty} \right\} \quad\text{and}\quad \mathbb{E}_{\leq 1} \;=\; \{\, Y \mid Y\colon \Sigma \to [0,\,1] \,\}\,.$$

An expectation X maps every program state to a non-negative real number or ∞. $\mathbb{E}_{\leq 1}$ allows for reasoning about probabilities of events whereas $\mathbb{E}$ allows for reasoning about expected values of more general random variables such as the expected value of a variable x, the expected height of a tree (in the heap), etc. Notice that a predicate is a particular expectation, namely its Iverson bracket, that maps only to $\{0,\ 1\}$. In contrast to [McIver and Morgan 2005], our expectations are not necessarily bounded. Hence, $(\mathbb{E},\ \preceq)$ and $(\mathbb{E}_{\leq 1},\ \preceq)$, where $X \preceq Y$ iff $\forall (s,\ h) \in \Sigma\colon X(s,\ h) \leq Y(s,\ h)$ each form a complete lattice with least element 0 and greatest element ∞ and 1, respectively.[3] We present most of our results with respect to the domain $(\mathbb{E},\ \preceq)$, i.e. we develop a logic for reasoning about expected values. A logic for reasoning about probabilities of events can be constructed analogously by using the complete lattice $(\mathbb{E}_{\leq 1},\ \preceq)$ instead.

Analogously to [Reynolds 2002], we call an expectation $X \in \mathbb{E}$ *domain-exact* iff for all stacks $s \in \mathcal{S}$ and heaps $h, h' \in \mathcal{H}$, $X(s,\ h) > 0$ and $X(s,\ h') > 0$ together implies that $\mathrm{dom}\,(h) = \mathrm{dom}\,(h')$, i.e. for a fixed stack, the domain of all heaps such that the quantity X does not vanish is constant.

We next lift the atomic formulas of SL to a quantitative setting: The *empty-heap predicate* $[\mathbf{emp}]$, which evaluates to 1 iff the heap is empty, is defined as

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

The *points-to predicate* $[e \mapsto e']$, evaluating to 1 iff the heap consists of exactly one cell with address e and content e', is defined as

$$[e \mapsto e'] \;=\; \lambda(s,h)\text{.}\begin{cases} 1, & \text{if } \mathrm{dom}\,(h) = \{s(e)\} \text{ and } h(s(e)) = s(e') \\ 0, & \text{otherwise.} \end{cases}$$

[3] By slight abuse of notation, for any constant $k \in \mathbb{R}_{\geq 0}^{\infty}$, we write k for $\lambda(s,\ h)\text{.}\ k$.

RIGHTSLINK

Notice that if $s(e) \notin \mathbb{N}_{>0}$ then automatically $\mathrm{dom}(h) \neq \{s(e)\}$. As a shorthand, we denote by $[e \mapsto e'_1, \ldots, e'_n]$ the predicate that evaluates to 1 on (s, h) iff the heap h contains exactly n cells with addresses $s(e)$, ..., $(e) + n - 1$ and respective contents $s(e'_1)$, ..., $s(e'_n)$.

The *allocated pointer predicate* $[e \mapsto -]$, which evaluates to 1 iff the heap consists of a single cell with address e (but arbitrary content), is defined as

$$[e \mapsto -] \;=\; \lambda(s,h)\boldsymbol{.}\; \begin{cases} 1, & \text{if } \mathrm{dom}(h) = \{s(e)\}, \\ 0, & \text{otherwise.} \end{cases}$$

All of the above predicates are domain-exact expectations evaluating to either zero or one.

As an example of a truly *quantitative* expectation consider the *heap size quantity*

$$\mathbf{size} \;=\; \lambda(s,h)\boldsymbol{.}\; |\mathrm{dom}(h)|,$$

where $|\mathrm{dom}(h)|$ denotes the cardinality of $\mathrm{dom}(h)$, which measures the number of allocated cells in a heap h. In contrast to the standard SL predicates, **size** is neither domain-exact nor a predicate.

3.2 Separating Connectives between Expectations

We now develop quantitative versions of SL's connectives. *Standard conjunction* ($\wedge$) is modeled by pointwise multiplication. This is backward compatible as for any two predicates φ and ψ we have $[\varphi \wedge \psi] = [\varphi] \cdot [\psi] = \lambda(s,\ h)\boldsymbol{.}\ [\varphi](s,\ h) \cdot [\psi](s,\ h)$. Towards a *quantitative separating conjunction*, let us first examine the classical case, which is defined for two predicates φ and ψ as

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

In words, a state (s,h) satisfies $\varphi \star \psi$ iff there exists a *partition* of the heap h into two heaps h_1 and h_2 such that the stack s together with heap h_1 satisfies φ, and s together with h_2 satisfies ψ.

How should we connect two expectations X and Y in a similar fashion? As logical "and" corresponds to a multiplication, we need to find a partition of the heap h into $h_1 \star h_2$, measure X in h_1, measure Y in h_2, and finally multiply these two measured quantities. The naive approach,

$$\big(X \star Y\big)(s,\ h) \quad = \quad \exists\, h_1, h_2\colon\ [h = h_1 \star h_2] \cdot X(s,h_1) \cdot Y(s,h_2),$$

is not meaningful. At the very least, it is ill-typed. Moreover, what precisely determined quantity would the above express? After all, the existentially quantified partition of h need not be unique.

Our key redemptive insight here is that $\exists$ should correspond to max. From an algebraic perspective, this corresponds to the usual interpretation of existential quantifiers in a complete Heyting algebra or Boolean algebra as a disjunction (cf. [Scott 2008] for an overview), which we will interpret as a maximum in the realm of expectations. In first-order logic, the effect of the quantified predicate $\exists v\colon \varphi(v)$ is so-to-speak to "maximize the truth of $\varphi(v)$" by a suitable choice of v. In QSL, instead of truth, we maximize a quantity: Out of all partitions $h = h_1 \star h_2$, we choose the one—out of finitely many for any given h—that maximizes the product $X(s,h_1) \cdot Y(s,h_2)$. We thus define the quantitative $\star$ as follows:

Definition 3.1 (Quantitative Separating Conjunction). The *quantitative separating conjunction* $X \star Y$ of two expectations $X, Y \in \mathbb{E}$ is defined as

$$X \star Y \;=\; \lambda(s,h)\boldsymbol{.}\; \max_{h_1,h_2} \left\{ X(s,h_1) \cdot Y(s,h_2) \;\middle|\; h = h_1 \star h_2 \right\}. \qquad \triangle$$

As a first sanity check, notice that this definition is backward compatible to the qualitative setting: For predicates φ and ψ, we have $([\varphi] \star [\psi])(s,h) \in \{0,1\}$ and moreover $([\varphi] \star [\psi])(s,h) = 1$ holds in QSL if and only if $(s,h) \models \varphi \star \psi$ holds in SL.

RIGHTSLINK

Next, we turn to *separating implication.* For SL, this is defined for predicates φ and ψ as

$$(s,h) \models \varphi \mathrel{-\!\!\star} \psi \quad \text{iff} \quad \forall h'\colon \quad h' \perp h \text{ and } (s,h') \models \varphi \text{ implies } (s, h \star h') \models \psi .$$

So (s,h) satisfies $\varphi \mathrel{-\!\!\star} \psi$ iff the following holds: Whenever we can find a heap h' disjoint from h such that stack s together with heap h' satisfies φ, then s together with the *conjoined* heap $h \star h'$ must satisfy ψ. In other words: We measure the truth of ψ in *extended* heaps $h \star h'$, where all admissible extensions h' must satisfy φ.

How should we connect expectations Y and X in a similar fashion? Intuitively, $Y \mathrel{-\!\!\star} X$ intends to measure X in extended heaps, subject to the fact that the extensions satisfy Y. Since the least element of our complete lattice, i.e. 0, corresponds to false when evaluating a predicate, we interpret *satisfying an expectation* Y as measuring some positive quantity, i.e. $Y(s,h) > 0$.

As for the universal quantifier, our key insight is now that—dually to $\exists$ corresponding to max—$\forall$ should correspond to min: Whereas in first-order logic the predicate $\forall v\colon \varphi(v)$ "minimizes the truth of $\varphi(v)$" by requiring that $\varphi(v)$ must be true for all choices of v, in QSL we minimize a quantity: Out of all heap extensions h' disjoint from h that satisfy a given expectation Y, we choose an extension that minimizes the quantity $X(s, h \star h')$. Intuitively speaking, we pick the smallest possible[4] extension h' that barely satisfies Y. Since for given Y and h, there may be infinitely many (or no) admissible choices for h', we define the quantitative $\mathrel{-\!\!\star}$ by an infimum:

$$Y \mathrel{-\!\!\star} X \;=\; \lambda(s,h)\text{.}\; \inf_{h'} \left\{ \frac{X(s, h \star h')}{Y(s,h')} \;\middle|\; h' \perp h \text{ and } Y(s,h') > 0 \right\} .$$

This definition is well-behaved with $(\mathbb{E}_{\leq 1}, \preceq)$ as the underlying lattice.[5] However, for the domain $(\mathbb{E}, \preceq)$ the above definition of $\mathrel{-\!\!\star}$ is not well-defined if $Y(s,h') = \infty$ holds. We thus restrict Y to predicates. The above definition then simplifies as follows:

Definition 3.2 (Quantitative Separating Implication). The *quantitative separating implication* $[\varphi] \mathrel{-\!\!\star} X$ of predicate φ and expectation $X \in \mathbb{E}$ is defined as

$$[\varphi] \mathrel{-\!\!\star} X \;=\; \lambda(s,h)\text{.}\; \inf_{h'} \;\{ X(s, h \star h') \mid h' \perp h \text{ and } (s,h') \models \varphi \} . \qquad \triangle$$

Unfortunately, backward compatibility for quantitative separating implication comes with certain reservations: Suppose for a particular state $(s,\ h)$ there exists no heap extension h' such that $(s,\ h') \models \varphi$. Then $\{ X(s, h \star h') \mid h' \perp h \text{ and } (s,h') \models \varphi \}$ is empty, and the greatest lower bound (within our domain $\mathbb{R}^{\infty}_{\geq 0}$) of the empty set is ∞ and not 1. In particular, false $\mathrel{-\!\!\star} \psi \equiv$ true holds in SL, but $0 \mathrel{-\!\!\star} [\psi] = \infty$ holds in QSL. Since $0 = [\text{false}]$ but $\infty \neq [\text{true}]$, backward compatibility of quantitative separating implication breaks here. As a silver lining, however, we notice that true is the greatest element in the complete lattice of predicates and correspondingly ∞ is the greatest element in $\mathbb{E}$. In this light, the above appears not at all surprising. In fact, if we restrict ourselves to the domain $(\mathbb{E}_{\leq 1}, \preceq)$ to reason about probabilities, we achieve full backward compatibility. To be precise, let us explicitly embed classical separation logic (SL) into QSL.

Definition 3.3 (Embedding of SL *into* QSL*).* Formulas in classical separation logic (SL) are embedded into quantitative separation logic by a function $\mathtt{qsl}[\![.]\!]\colon \mathsf{SL} \to \mathbb{E}_{\leq 1}$ mapping formulas in SL to expectations in $\mathbb{E}_{\leq 1}$. This function is defined inductively as follows:

$$\mathtt{qsl}[\![\varphi]\!] \;=\; [\varphi] \text{ for any atomic formula } \varphi \in \mathsf{SL} \qquad \mathtt{qsl}[\![\neg\varphi]\!] \;=\; 1 - \mathtt{qsl}[\![\varphi]\!]$$

$$\mathtt{qsl}[\![\varphi_1 \star \varphi_2]\!] \;=\; \mathtt{qsl}[\![\varphi_1]\!] \star \mathtt{qsl}[\![\varphi_2]\!] \qquad \mathtt{qsl}[\![\varphi_1 \mathrel{-\!\!\star} \varphi_2]\!] \;=\; \mathtt{qsl}[\![\varphi_1]\!] \mathrel{-\!\!\star} \mathtt{qsl}[\![\varphi_2]\!]$$

[4] In terms of measuring $X(s,\ h \star h')$.

[5] In particular, quantitative separating implication and quantitative separating conjunction are adjoint.

RIGHTSLINK

$$\texttt{qsl}[\![\exists x\colon \varphi]\!] \;=\; \sup_{v\in\mathbb{Z}} \texttt{qsl}[\![\varphi]\!]\,[x/v] \qquad \texttt{qsl}[\![\varphi_1 \wedge \varphi_2]\!] \;=\; \texttt{qsl}[\![\varphi_1]\!]\cdot\texttt{qsl}[\![\varphi_2]\!] \qquad \triangle$$

Every atomic separation logic formula is thus interpreted as its Iverson bracket in QSL. Furthermore, every connective is replaced by its quantitative variant. We then obtain that QSL—as an assertion language—is a conservative extension of classical separation logic.

Theorem 3.4 (Conservativity of QSL as an assertion language). *For all classical separation logic formulas $\varphi \in \mathsf{SL}$ and all states $(s,h) \in \Sigma$, we have*

(1) $\texttt{qsl}[\![\varphi]\!](s,h) \in \{0,1\}$, *and*
(2) $(s,h) \models \varphi$ *if and only if* $\texttt{qsl}[\![\varphi]\!](s,h) \;=\; 1$.

The same result is achieved for the expectation domain $(\mathbb{E}, \preceq)$ if we define the embedding of separating implication as $\texttt{qsl}[\![\varphi_1 \mathrel{-\!\!\star} \varphi_2]\!] = \min\{1, \texttt{qsl}[\![\varphi_1]\!] \mathrel{-\!\!\star} \texttt{qsl}[\![\varphi_2]\!]\}$.

3.3 Properties of Quantitative Separating Connectives

Besides backward compatibility, the separating connectives of QSL are well-behaved in the sense that they satisfy most properties of their counterparts in SL. To justify this claim, we now present a collection of quantitative analogs of properties of classical separating conjunction and implication. Most of those properties originate from the seminal papers on classical separation logic [Ishtiaq and O'Hearn 2001; Reynolds 2002]. We start with algebraic laws for quantitative separating conjunction:

Theorem 3.5. $(\mathbb{E}, \star, [\mathbf{emp}])$ *is a commutative monoid, i.e. for all $X, Y, Z \in \mathbb{E}$ the following holds:*

(1) Associativity: $X \star (Y \star Z) \;=\; (X \star Y) \star Z$
(2) Neutrality of $[\mathbf{emp}]$: $X \star [\mathbf{emp}] \;=\; [\mathbf{emp}] \star X \;=\; X$
(3) Commutativity: $X \star Y \;=\; Y \star X$

Theorem 3.6 ((Sub)distributivity Laws). *Let $X, Y, Z \in \mathbb{E}$ and let φ be a predicate. Then:*

(1) $X \star \max\{\, Y,\; Z\,\} \;=\; \max\{\, X \star Y,\; X \star Z\,\}$
(2) $X \star (Y + Z) \;\preceq\; X \star Y + X \star Z$
(3) $[\varphi] \star (Y \cdot Z) \;\preceq\; \big([\varphi] \star Y\big) \cdot \big([\varphi] \star Z\big)$

Furthermore, if X and $[\varphi]$ are domain-exact, we obtain full distributivity laws:

(4) $X \star (Y + Z) \;=\; X \star Y + X \star Z$
(5) $[\varphi] \star (Y \cdot Z) \;=\; \big([\varphi] \star Y\big) \cdot \big([\varphi] \star Z\big)$

The max in Theorem 3.6.1 corresponds to a disjunction ($\vee$) in the classical setting as for any two predicates φ and ψ we have $[\varphi \vee \psi] = \max\{\,[\varphi],\; [\psi]\,\}$, where the max is taken pointwise. Moreover, if φ and ψ are mutually exclusive, i.e. $[\varphi]\cdot[\psi] = 0$, their maximum coincides with their sum. That is, we have $\max\{[\varphi], [\psi]\} = [\varphi] + [\psi]$. Theorem 3.6.1 shows that $\star$ distributes over max. Unfortunately, for $+$ we only have sub-distributivity (Theorem 3.6.2). We recover full distributivity in case that X is domain-exact (Theorem 3.6.4).

A further important analogy to SL is that quantitative separating conjunction is monotonic:

Theorem 3.7 (Monotonicity of $\star$). *$X \preceq X'$ and $Y \preceq Y'$ implies $X \star Y \preceq X' \star Y'$.*

Next, we look at a quantitative analog to *modus ponens*. The classical modus ponens rule states that $\varphi \star (\varphi \mathrel{-\!\!\star} \psi)$ implies ψ. In a quantitative setting, implication generalizes to $\preceq$, i.e. the partial order we defined in Section 3.1 (see also [McIver and Morgan 2005]).

Theorem 3.8 (Quantitative Modus Ponens). $[\varphi] \star \big([\varphi] \mathrel{-\!\!\star} X\big) \;\preceq\; X$.

Analogously to the qualitative setting, quantitative $\star$ and $\mathrel{-\!\!\star}$ are adjoint operators:

RIGHTSLINK

Theorem 3.9 (Adjointness of $\star$ and $\mathrel{-\!\!\star}$). $X \star [\varphi] \;\preceq\; Y \quad \text{iff} \quad X \;\preceq\; [\varphi] \mathrel{-\!\!\star} Y.$

Intuitively, a separating conjunction $_ \star [\varphi]$ carves out a portion of the heap, since $X \star [\varphi]$ splits of a part of the heap satisfying φ and measures X in the remaining heap. Conversely, $[\varphi] \mathrel{-\!\!\star} _$ extends the heap by a portion satisfying φ. Adjointness now tells us that instead of carving out something on the left-hand side of an inequality, we can extend something on the right-hand side and vice versa. This is analogous to $a - \epsilon \leq b$ iff $a \leq \epsilon + b$ in standard calculus.

Example 3.10. Let us consider a few examples to gain more intuition on quantitative separating connectives. For that, let s be any stack and let heap $h = \{1 \mapsto 2, 2 \mapsto 3, 4 \mapsto 5\}$. Then:

$$\begin{aligned}
([1 \mapsto 2] \star \mathbf{size})\,(s,h) &= 2 = \mathbf{size}(s,h) - 1 \\
([3 \mapsto 4] \mathrel{-\!\!\star} \mathbf{size})\,(s,h) &= 4 = \mathbf{size}(s,h) + 1 \\
([3 \mapsto 4] \star \mathbf{size})\,(s,h) &= 0 = ([1 \mapsto 2] \star [1 \mapsto 2] \star \mathbf{size})\,(s,h) \\
([1 \mapsto 2] \star ([1 \mapsto 2] \mathrel{-\!\!\star} \mathbf{size}))\,(s,h) &= 3 = \mathbf{size}(s,h) = ([3 \mapsto 4] \mathrel{-\!\!\star} ([3 \mapsto 4] \star \mathbf{size}))\,(s,h) \\
([1 \mapsto 2] \mathrel{-\!\!\star} \mathbf{size})\,(s,h) &= \infty = ([3 \mapsto 4] \mathrel{-\!\!\star} ([3 \mapsto 4] \mathrel{-\!\!\star} \mathbf{size}))\,(s,h)
\end{aligned} \qquad \triangle$$

3.4 Pure Expectations

In SL, a predicate is called *pure* iff its truth does not depend on the heap but only on the stack. Analogously, in QSL we call an expectation X pure iff

$$\forall\, s, h_1, h_2\colon \quad X(s, h_1) \;=\; X(s, h_2)\,.$$

For pure expectations, several of [Reynolds 2002] laws for SL hold as well:

Theorem 3.11 (Algebraic Laws for $\star$ under Purity). *Let $X, Y, Z \in \mathbb{E}$ and let X be pure. Then*

(1) $X \cdot Y \preceq X \star Y$,
(2) $X \cdot Y \;=\; X \star Y$, *if additionally Y is also pure, and*
(3) $(X \cdot Y) \star Z = X \cdot (Y \star Z)$.

3.5 Intuitionistic Expectations

In SL, a predicate φ is called *intuitionistic*, iff for all stacks s and heaps h, h' with $h \subseteq h'$, $(s,\ h) \models \varphi$ implies $(s,\ h') \models \varphi$. So as we extend the heap from h to h', an intuitionistic predicate can only get *"more true"*. Analogously, in QSL, as we extend the heap from h to h', the quantity measured by an *intuitionistic expectation* can only *increase*. Formally, an expectation X is called *intuitionistic* iff

$$\forall\, s, h \subseteq h'\colon \quad X(s,h) \;\leq\; X(s,h')\,.$$

A natural example of an intuitionistic expectation is the heap size quantity

$$\mathbf{size} \;=\; \lambda(s,h)\boldsymbol{.}\; |\mathrm{dom}\,(h)\,|\,.$$

[Reynolds 2002] describes a systematic way to construct intuitionistic predicates from possibly non-intuitionistic ones: For any predicate φ, $\varphi \star \mathsf{true}$ is the strongest intuitionistic predicate weaker than φ, and $\mathsf{true} \mathrel{-\!\!\star} \varphi$ is the weakest intuitionistic predicate stronger than φ. In QSL:

Theorem 3.12 (Tightest Intuitionistic Expectations). *Let $X \in \mathbb{E}$. Then:*

(1) *$X \star 1$ is the smallest intuitionistic expectation that is greater than X. Formally, $X \star 1$ is intuitionistic, $X \preceq X \star 1$, and for all intuitionistic X' satisfying $X \preceq X'$, we have $X \star 1 \preceq X'$.*
(2) *$1 \mathrel{-\!\!\star} X$ is the greatest intuitionistic expectation that is smaller than X. Formally, $1 \mathrel{-\!\!\star} X$ is intuitionistic, $1 \mathrel{-\!\!\star} X \preceq X$, and for all intuitionistic X' satisfying $X' \preceq X$, we have $X' \preceq 1 \mathrel{-\!\!\star} X$.*

RIGHTSLINK

For example, the *contains-pointer predicate* $[e \hookrightarrow e']$ defined by

$$[e \hookrightarrow e'] \;=\; [e \mapsto e'] \star 1$$

is an intuitionistic version of the points-to predicate $[e \mapsto e']$: Whereas $[e \mapsto e']$ evaluates to 1 iff the heap consists of *exactly* one cell with value e' at address e and no other cells, $[e \hookrightarrow e']$ evaluates to 1 iff the heap *contains* a cell with value e' at address e but possibly also other allocated memory.

Analogously, the fact that some cell with address e exists on the heap is formalized by

$$[e \hookrightarrow -] \;=\; [e \mapsto -] \star 1 \,.$$

With intuitionistic versions of points-to predicates at hand, we can derive specialized laws when dealing with the heap size quantity, which we already observed for a concrete heap in Example 3.10.

Theorem 3.13 (Heap Size Laws). *Let $X, Y \in \mathbb{E}$ and e, e' be arithmetic expressions. Then:*

(1) $[e \mapsto e'] \star \textbf{size} \;=\; [e \hookrightarrow e'] \cdot (\textbf{size} - 1)$
(2) $[e \mapsto e'] \mathrel{-\!\!\star} \textbf{size} \;=\; 1 + \textbf{size} + [e \hookrightarrow -] \cdot \infty$
(3) $(X \star Y) \cdot \textbf{size} \;\preceq\; (X \cdot \textbf{size}) \star Y + X \star (Y \cdot \textbf{size})$
(4) $(X \star Y) \cdot \textbf{size} \;=\; (X \cdot \textbf{size}) \star Y + X \star (Y \cdot \textbf{size})$, *if X or Y is domain-exact.*

The first two rules illustrate the role of $\star$ and $\mathrel{-\!\!\star}$: $\star$ removes a part of the heap that is measured and consequently decreases the size of the remaining heap. Dually, $\mathrel{-\!\!\star}$ extends the heap and hence increases its size. If the heap cannot be extended appropriately, the infimum in the definition of $\mathrel{-\!\!\star}$ yields ∞. The third and fourth rule intuitively state that the size of the heap captured by $X \star Y$ is the sum of the sizes of the heap captured by X, i.e. $X \cdot \textbf{size}$, and of the heap captured by Y, i.e. $Y \cdot \textbf{size}$. However, in both cases we have to account for parts of the heap whose size is not measured, i.e. Y if we measure the size of X and vice versa. These parts are "absorbed" by an additional separating conjunction with Y and X, respectively.

3.6 Recursive Expectation Definitions

To reason about unbounded data structures such as lists, trees, etc., separation logic relies on inductive predicate definitions (cf. [Brotherston 2007; Reynolds 2002]). In QSL, quantitative properties of unbounded data structures are specified similarly using recursive equations of the form

$$P(\vec{\alpha}) \;=\; X_P(\vec{\alpha}), \tag{1}$$

where $\vec{\alpha} \in \mathbb{Z}^n$, $P\colon \mathbb{Z}^n \to \mathbb{E}$, and $X.(\,\cdot\,)\colon (\mathbb{Z}^n \to \mathbb{E}) \to (\mathbb{Z}^n \to \mathbb{E})$ is a monotone function.

Example 3.14. Consider a recursive predicate definition from standard separation logic: A singly-linked list segment with head α and tail β is given by the equation

$$[\mathsf{ls}\,(\alpha,\beta)] \;=\; \underbrace{[\alpha=\beta]\cdot[\textbf{emp}] + [\alpha\neq\beta]\cdot \sup\nolimits_{\gamma}\; [\alpha\mapsto\gamma]\star[\mathsf{ls}\,(\gamma,\beta)]}_{=: X_{\mathsf{ls}}(\alpha,\beta)}\,.$$

Clearly, $X_P(\alpha,\beta)$ is monotone, i.e. $P \preceq P'$ implies $X_P(\alpha,\beta) \preceq X_{P'}(\alpha,\beta)$. Hence, all list segments between α and β are given by the least fixed point of the above equation. △

The semantics of (1) is defined as the least fixed point of a monotone expectation transformer

$$\Psi_P\colon \quad (\mathbb{Z}^n \to \mathbb{E}) \to (\mathbb{Z}^n \to \mathbb{E}), \quad Q \mapsto \lambda\vec{\alpha}.\, X_Q(\vec{\alpha}).$$

Thus, we define the expectation given by recursive equation (1) as $P(\vec{\alpha}) = \big(\mathsf{lfp}\; Q.\, \Psi_P(Q)\big)(\vec{\alpha})$, where $\mathsf{lfp}\; Q.\, \Psi(Q)$ denotes the least fixed point of Ψ. Existence of the least fixed point is guaranteed due to Tarski and Knaster's fixed point theorem (cf. [Cousot and Cousot 1979]).

RIGHTSLINK

This notion of recursive definitions coincides with the semantics of inductive predicates in SL [Brotherston 2007] if expectations are restricted to predicates. For instance, $[\mathsf{ls}(\alpha, \beta)](s, h) = 1$ iff h consists *exactly* of a singly-linked list with head α and tail β.

Recursive expectation definitions in QSL are, however, not limited to predicates. For example, the *length* of a singly-linked list segment can be defined as follows:

$$\mathsf{len}(\alpha, \beta) \quad = \quad [\alpha \neq \beta] \cdot \sup_{\gamma} \; [\alpha \mapsto \gamma] \star ([\mathsf{ls}(\gamma, \beta)] + \mathsf{len}(\gamma, \beta))$$

If the heap exclusively consists of a singly-linked list from α to β, then the expectation $\mathsf{len}(\alpha, \beta)$ evaluates to the length of that list, and to zero otherwise. We next collect a few properties of the two closely related expectations len and ls that simplify reasoning about programs.

Lemma 3.15 (Properties of List Segments and Lengths of List Segments). *We have:*

(1) $\mathsf{len}(\alpha, \beta) \quad = \quad [\mathsf{ls}(\alpha, \beta)] \cdot \mathbf{size}$
(2) $[\mathsf{ls}(\alpha, \beta)] \quad = \quad \sup_{\gamma} [\mathsf{ls}(\alpha, \gamma)] \star [\mathsf{ls}(\gamma, \beta)]$

The first property gives an alternative characterization of list lengths which exploits the fact that $[\mathsf{ls}]$ ensures that nothing but a list is contained in the heap. Consequently, the length of that list is given by the size of the specified heap. The second property shows that lists can be split into multiple lists or merged into a single list at any address in between.

The list-length quantity len actually serves two purposes: It ensures that the heap is a list and if so determines the longest path through the heap. The latter part can be generalized to other data structures. To this end, assume the heap is organized into fixed-size, successive blocks of memory representing *records*, for example the left and right pointer of a binary tree. If the size of records is a constant $n \in \mathbb{N}$, then the longest path through these records starting in α is given by

$$\mathsf{path}[\![n]\!](\alpha) \quad = \quad \sup_{\beta \in \mathbb{N}} \Big((\max_{0 \leq k < n} [\alpha + k \mapsto \beta]) \star \big(1 + \mathsf{path}[\![n]\!](\beta)\big) \Big) \; .$$

Intuitively, $\mathsf{path}[\![n]\!](\alpha)$ always selects the successor address β among the possible pointers in the record belonging to α which is the source of the longest path through the remaining heap. Notice that no explicit base case is needed, because the length of empty paths is zero. Moreover, the use of the separating conjunction prevents selecting the same pointer twice. The quantity path is more liberal than len in the sense that heaps may contain pointers that do not lie on the specified path. The path quantity can then be easily combined with stricter data structure specifications.

Example 3.16. Consider a classical recursive SL predicate specifying binary trees with root α:

$$[\mathsf{tree}(\alpha)] \quad = \quad [\alpha = 0] \cdot [\mathbf{emp}] \; + \; \sup_{\beta, \gamma \in \mathbb{N}} \; [\alpha \mapsto \beta, \gamma] \star [\mathsf{tree}(\beta)] \star [\mathsf{tree}(\gamma)] \; .$$

Combining $[\mathsf{tree}(\alpha)]$ with $\mathsf{path}[\![2]\!](\alpha)$, we can measure the height of binary trees with root α:

$$\mathsf{treeHeight}(\alpha) \quad = \quad [\mathsf{tree}(\alpha)] \cdot \mathsf{path}[\![2]\!](\alpha) \; .$$

This is illustrated in Figure 3, where two heaps are graphically depicted as directed graphs. The left graph contains a cycle and thus does *not* constitute a binary tree. Consequently, $[\mathsf{tree}(\alpha)] = 0$. The longest path through this heap is $\alpha\beta_1 \ldots \beta_4$, i.e. $\mathsf{path}[\![2]\!](\alpha) = 5$. In contrast, the right graph *is* a binary tree with root α, i.e. $[\mathsf{tree}(\alpha)] = 1$. The longest path through this heap is of length two, e.g. $\alpha\beta_1\beta_2$. Hence, the height of the tree is given by $\mathsf{treeHeight}(\alpha) = [\mathsf{tree}(\alpha)] \cdot \mathsf{path}[\![2]\!](\alpha) = 2$. △

RIGHTSLINK

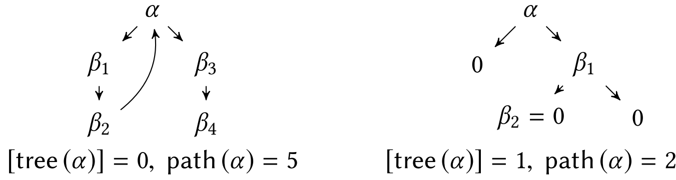

Fig. 3. Evaluation of $[\mathrm{tree}\,(\alpha)]$ and $\mathrm{path}\,(\alpha)$ for two heaps depicted as graphs. Here, an edge $x \longrightarrow y$ denotes $h(s(x)) = s(y)$ or $h(s(x+1)) = s(y)$.

4 REASONING ABOUT PROGRAMS

We now turn from QSL as an assertion language to program verification. Classical separation logic is commonly applied as a basis for Floyd-Hoare-style correctness proofs. The main concept in Floyd-Hoare logic are *Hoare triples*. A Hoare triple $\langle\, \varphi \,\rangle\; c\; \langle\, \psi \,\rangle$ consists of a precondition φ, a non–probabilistic program c, and a postcondition ψ.

One approach to proving a triple $\langle\, \varphi \,\rangle\; c\; \langle\, \psi \,\rangle$ valid is to determine whether precondition φ is covered by all initial states that — executed on c — reach a final state satisfying postcondition ψ. This kind of *backward reasoning* corresponds to Dijkstra's weakest preconditions. More precisely, the *weakest precondition of* c *with respect to postcondition* ψ is the weakest predicate $\mathrm{wp}\llbracket c \rrbracket\,(\psi)$, such that the triple $\left\langle\, \mathrm{wp}\llbracket c \rrbracket\,(\psi) \,\right\rangle\; c\; \left\langle\, \psi \,\right\rangle$ is valid, i.e. $\mathrm{wp}\llbracket c \rrbracket\,(\psi)$ is the predicate such that

$$\forall\, \varphi: \qquad \varphi \implies \mathrm{wp}\llbracket c \rrbracket\,(\psi) \quad \text{iff} \quad \langle\, \varphi \,\rangle\; c\; \langle\, \psi \,\rangle \text{ is valid}\,.$$

For SL, validity of Hoare triples usually includes that "correct programs do not fail" [Reynolds 2002; Yang and O'Hearn 2002], i.e. no execution satisfying the precondition may lead to a memory fault.

Reasoning about probabilistic programs is more subtle. Running a probabilistic program on an initial state does not yield one or more final states, but a *sub*distribution of final states. The missing probability mass corresponds to the *probability of nontermination or encountering a memory fault*. Furthermore, when performing *quantitative* reasoning, the notion of correctness becomes blurred. For instance, it might be acceptable that a program fails with some small probability.

In order to account for probabilistic behavior, [Kozen 1983] generalized weakest precondition reasoning from predicates to measurable functions and later [McIver and Morgan 2005] (re)introduced nondeterminism and coined the term *weakest preexpectation*. To incorporate dynamic memory, we extend their approach by lifting the backward reasoning rules of [Ishtiaq and O'Hearn 2001; Reynolds 2002] to a quantitative setting. To be precise, our calculus is designed for *total correctness*, asserts that *no memory faults* happen during any execution (with positive probability), and assumes a *demonic* interpretation of nondeterminism. Alternative design choices are discussed in Section 5.

Notice that forward reasoning in the sense of strongest postexpectations is not an option as in general strongest postexpectations do not exist for probabilistic programs [Jones 1990]. This also justifies our need for the separating implication in QSL which — in classical approaches based on separation logic — is not needed when applying forward reasoning.

4.1 Weakest Preexpectations

The *weakest preexpectation* of program c with respect to *postexpectation* $X \in \mathbb{E}$ is an expectation $\mathrm{wp}\llbracket c \rrbracket\,(X) \in \mathbb{E}$, such that $\mathrm{wp}\llbracket c \rrbracket\,(X)\,(s,\, h)$ is the *least expected value* of X (measured in the final states) *after successful termination*, i.e. *no memory faults* during execution, of c on initial state $(s,\, h)$. In particular, if X is a predicate then $\mathrm{wp}\llbracket c \rrbracket\,(X)\,(s, h)$ is the least probability that c executed on initial state $(s,\, h)$ does not cause a memory fault and terminates successfully in a final state satisfying X.

RIGHTSLINK

Table 1. Rules for the weakest preexpectation transformer. Here $X \in \mathbb{E}$ is a (post)expectation, $X[x/v] = \lambda(s,h)\centerdot X(s[x/s(v)],h)$ is the "syntactic replacement" of x by v in X, and $\vec{e} = (e_1, \ldots, e_n)$ is a tuple of expressions. Moreover, lfp $Y\centerdot \Phi(Y)$ is the least fixed point of Φ.

c	$\mathsf{wp}[\![c]\!](X)$
`skip`	X
x := e	$X[x/e]$
$c_1 ; c_2$	$\mathsf{wp}[\![c_1]\!]\big(\mathsf{wp}[\![c_2]\!](X)\big)$
`if` (b) $\{c_1\}$ `else` $\{c_2\}$	$[b]\cdot \mathsf{wp}[\![c_1]\!](X) + [\neg b]\cdot \mathsf{wp}[\![c_2]\!](X)$
`while` (b) $\{c'\}$	$\mathsf{lfp}\, Y\centerdot\ [\neg b]\cdot X + [b]\cdot \mathsf{wp}[\![c']\!](Y)$
$\{c_1\}\,[p]\,\{c_2\}$	$p\cdot \mathsf{wp}[\![c_1]\!](X) + (1-p)\cdot \mathsf{wp}[\![c_2]\!](X)$
x := new$(\vec{e})$	$\inf_{v\in\mathbb{N}_{>0}} [v \mapsto \vec{e}] \mathbin{-\!\!\star} X[x/v]$
x := <e>	$\sup_{v\in\mathbb{Z}} [e \mapsto v] \star \big([e \mapsto v] \mathbin{-\!\!\star} X[x/v]\big)$
<e> := e'	$[e \mapsto -] \star \big([e \mapsto e'] \mathbin{-\!\!\star} X\big)$
`free`(e)	$[e \mapsto -] \star X$

In the following, we extend the weakest preexpectation calculus of [McIver and Morgan 2005] to heap-manipulating programs, i.e. hpGCL as presented in Section 2.

Definition 4.1 (Weakest Preexpectation Transformer). The *weakest preexpectation* $\mathsf{wp}[\![c]\!](X)$ *of* $c \in$ hpGCL *with respect to postexpectation* $X \in \mathbb{E}$ is defined according to the rules in Table 1. △

Let us go over the individual rules for wp stated in Table 1. We start with briefly considering the non-heap-manipulating constructs. $\mathsf{wp}[\![\texttt{skip}]\!]$ behaves as the identity since `skip` does not modify the program state. For $\mathsf{wp}[\![x := e]\!](X)$ we return $X[x/e]$ which is obtained from X by "syntactically replacing" x with e. More formally, $X[x/e] = \lambda(s,h)\centerdot X(s[x/s(e)],h)$. For sequential composition, $\mathsf{wp}[\![c_1 ; c_2]\!](X)$ obtains a preexpectation of the program $c_1 ; c_2$ by applying $\mathsf{wp}[\![c_1]\!]$ to the intermediate expectation obtained from $\mathsf{wp}[\![c_2]\!](X)$. For conditional choice, $\mathsf{wp}[\![\texttt{if}\ (b)\ \{c_1\}\ \texttt{else}\ \{c_2\}]\!](X)$ selects either $\mathsf{wp}[\![c_1]\!](X)$ or $\mathsf{wp}[\![c_2]\!](X)$ by multiplying them accordingly with the indicator function of b or the indicator function of $\neg b$ and adding those two products. For the probabilistic choice, $\mathsf{wp}[\![\{c_1\}\,[p]\,\{c_2\}]\!](X)$ is a convex sum that weighs $\mathsf{wp}[\![c_1]\!](X)$ and $\mathsf{wp}[\![c_2]\!](X)$ by probabilities p and $(1-p)$, respectively. For loops, $\mathsf{wp}[\![\texttt{while}\,(b)\,\{c'\}]\!](X)$ is characterized as a least fixed point of loop unrollings. We discuss loops and corresponding proof rules in Section 4.4. For a detailed treatment of weakest preexpectations for these standard constructs, confer [McIver and Morgan 2005]. Before we consider the remaining statements, let us collect a few basic properties of wp:

THEOREM 4.2 (BASIC PROPERTIES OF wp). *For all* hpGCL*-programs* c*, expectations* $X, Y \in \mathbb{E}$*, predicates* φ *and constants* $k \in \mathbb{R}_{\geq 0}$*, we have:*

(1) *Monotonicity:* $X \preceq Y$ implies $\mathsf{wp}[\![c]\!](X) \preceq \mathsf{wp}[\![c]\!](Y)$
(2) *Super–linearity:* $\mathsf{wp}[\![c]\!](k\cdot X + Y) \preceq k\cdot \mathsf{wp}[\![c]\!](X) + \mathsf{wp}[\![c]\!](Y)$
(3) *Strictness:* $\mathsf{wp}[\![c]\!](0) = 0$
(4) 1*–Boundedness of Predicates:* $\mathsf{wp}[\![c]\!]([\varphi]) \preceq 1.$

Additionally, if c *does not contain an allocation statement* x := new$(\vec{e})$*, we have:*

RIGHTSLINK

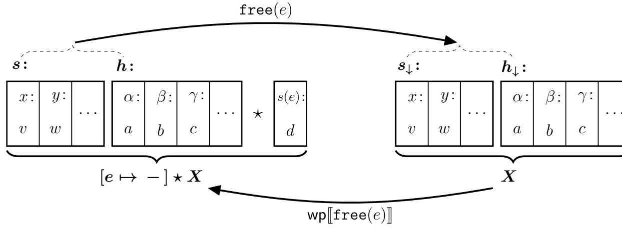

Fig. 4. Weakest preexpectation of memory deallocation.

(5) *ω-continuity: For every increasing ω-chain $X_1 \preceq X_2 \preceq \ldots$ in $\mathbb{E}$, we have*

$$\sup_n \mathsf{wp}[\![c]\!]\,(X_n) \;=\; \mathsf{wp}[\![c]\!]\,\big(\sup_n X_n\big) \;.$$

(6) *Linearity:* $\mathsf{wp}[\![c]\!]\,(k \cdot X + Y) \;=\; k \cdot \mathsf{wp}[\![c]\!]\,(X) + \mathsf{wp}[\![c]\!]\,(Y)$

4.2 Deallocation, Heap Mutation, and Lookup

We now go over the definitions for deterministic heap–accessing language constructs in Table 1.

Memory deallocation. A memory cell is deleted from the current heap using the $\mathtt{free}(e)$ construct as illustrated in Figure 4. $\mathtt{free}(e)$ starts on some initial state $(s,\, h)$ shown on the left-hand side and tries to deallocate the memory cell with address $s(e)$. In case that $s(e)$ is a valid address (as depicted in Figure 4), i.e. $s(e) \in \mathsf{dom}\,(h)$, $\mathtt{free}(e)$ removes the corresponding cell from the heap and terminates in a final state $(s_\downarrow,\, h_\downarrow)$ shown on the right-hand side. In case that $s(e)$ is not a valid address (*not* depicted in Figure 4), i.e. $s(e) \notin \mathsf{dom}\,(h)$, $\mathtt{free}(e)$ crashes.

What is the weakest preexpectation of $\mathtt{free}(e)$ with respect to a postexpectation X? For answering that, we need to construct an expectation $\mathsf{wp}[\![\mathtt{free}(e)]\!]\,(X)$, such that the quantity $\mathsf{wp}[\![\mathtt{free}(e)]\!]\,(X)$ measured in the initial state coincides with quantity X measured in the final state. The way we will construct $\mathsf{wp}[\![\mathtt{free}(e)]\!]\,(X)$ is to *measure X in the initial state* and successively rectify the difference to *measuring X in the final state*. So what is that difference? We need to dispose the allocated memory cell with address $s(e)$ in the initial state. We can rectify this through (a) ensuring that this memory cell actually exists and (b) notionally separating it from the rest of the heap and measuring X only in that rest. Both (a) and (b) are achieved by separatingly conjoining X with $[e \mapsto -]$, thus obtaining $[e \mapsto -] \star X$. Notice that only heaps consisting of a single cell with address $s(e)$ make $[e \mapsto -]$ evaluate to 1 and are hence the only possible choices such that $[e \mapsto -] \star X$ is evaluated to some quantity possibly larger than 0. This also means that if $\mathtt{free}(e)$ crashes because address $s(e)$ is not allocated, then $\mathsf{wp}[\![\mathtt{free}(e)]\!]\,(X)\,(s,\, h)$ correctly yields 0.

Memory allocation. The memory allocation statement $x := \mathtt{new}\,(e)$ deserves special attention as it is the only statement that exhibits nondeterministic behavior. For simplicity, let us consider $x := \mathtt{new}\,(e)$ instead of $x := \mathtt{new}\,(\vec{e})$, i.e. we only allocate a *single* memory cell. The situation is illustrated in Figure 5. Operationally, the instruction $x := \mathtt{new}\,(e)$ starts on some initial state $(s,\, h)$ shown on the left-hand side, adds (allocates) to the domain of heap h a single *fresh* address v, and stores at this address content $s(e)$. After allocating memory at address v, the address v is stored in variable x. The statement $x := \mathtt{new}\,(e)$ then terminates in a final state $(s_\downarrow,\, h_\downarrow)$ shown on the right-hand side. Since v is chosen *nondeterministically*, we cannot give any a-priori guarantees on v except for $v \notin \mathsf{dom}\,(h)$. Furthermore, notice that in our memory model there are at any point infinitely many free addresses available for allocation. Allocation thus never causes a memory fault.

RIGHTSLINK

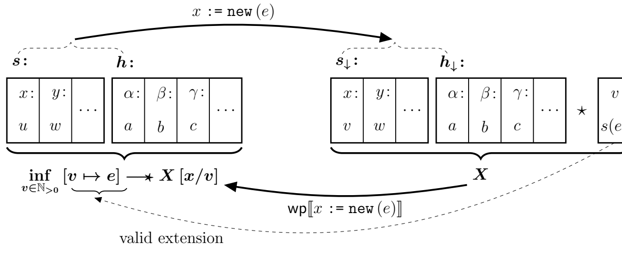

Fig. 5. Weakest preexpectation of memory allocation.

What is now the weakest preexpectation of $x := \texttt{new}(e)$ with respect to a postexpectation X? Again, we construct $\mathsf{wp}\llbracket x := \texttt{new}(e) \rrbracket (X)$ by measuring X in the initial state and rectifying the differences to measuring X in the final state. So what are those differences? The first difference is that we are missing in the initial state the newly allocated memory cell with address v and content $s(e)$ which is present in the final state. We can rectify this through notionally extending the heap of the initial state by measuring $[v \mapsto e] \mathrel{-\!\!\star} X$ instead of X. Notice that a heap consisting of a single cell with address v and content $s(e)$ is the *only* valid extension that satisfies $[v \mapsto e]$. The next difference is that in the final state variable x has value v. We can mimic this by a syntactic replacement of x by v in X, thus obtaining $[v \mapsto e] \mathrel{-\!\!\star} X[x/v]$. Finally, we have to account for the fact that the newly allocated address v is chosen nondeterministically. Following McIver and Morgan's demonic nondeterminism school of thought, we select by $\inf_{v \in \mathbb{N}_{>0}}$ any address that *minimizes* the sought-after quantity. We thus obtain $\mathsf{wp}\llbracket x := \texttt{new}(e) \rrbracket (X) = \inf_{v \in \mathbb{N}_{>0}} [v \mapsto e] \mathrel{-\!\!\star} X[x/v]$.

Heap mutation. Figure 6 illustrates how the heap is mutated by a statement $<e> := e'$. Operationally, we can dissect this instruction into two parts: Starting in some initial state (s, h) shown on the left-hand side, we first deallocate the memory at address $s(e)$ by $\texttt{free}(e)$ and thereby obtain an intermediate state (s', h'). Second, we allocate a new memory cell with content $s(e')$. In contrast to the statement $x := \texttt{new}(e')$, which is addressed in the next section, the address of that cell is fixed to $s(e)$. This is achieved by the instruction $\texttt{new}(e')\texttt{@}e$, which we introduce here ad-hoc just for illustration purposes. Consequently, the weakest preexpectation of $\texttt{new}(e')\texttt{@}e$ coincides with the weakest preexpectation of $x := \texttt{new}(e')$ except that (a) the allocated address v is fixed to $s(e)$ and (b) we do not perform an assignment to x. Thus, $\mathsf{wp}\llbracket \texttt{new}(e')\texttt{@}e \rrbracket (X) = [e \mapsto e'] \mathrel{-\!\!\star} X$.

Since $<e> := e'$ has the same effect as $\texttt{free}(e)\,;\, \texttt{new}(e')\texttt{@}e$, its weakest preexpectation is given by

$$
\begin{aligned}
\mathsf{wp}\llbracket <e> := e' \rrbracket (X) &= \mathsf{wp}\llbracket \texttt{free}(e)\,;\, \texttt{new}(e')\texttt{@}e \rrbracket (X) && \text{(see above)} \\
&= \mathsf{wp}\llbracket \texttt{free}(e) \rrbracket \big(\mathsf{wp}\llbracket \texttt{new}(e')\texttt{@}e \rrbracket (X)\big) && \text{(see Table 1)} \\
&= \mathsf{wp}\llbracket \texttt{free}(e) \rrbracket \left([e \mapsto e'] \mathrel{-\!\!\star} X\right) && \text{(see above)} \\
&= [e \mapsto -\,] \star \big([e \mapsto e'] \mathrel{-\!\!\star} X\big). && \text{(see Table 1)}
\end{aligned}
$$

Another explanation of $[e \mapsto -\,] \star \big([e \mapsto e'] \mathrel{-\!\!\star} X\big)$ from a syntactic point of view is as follows: By $[e \mapsto -\,] \star \ldots$, we ensure that the heap contains a cell with address e and carve it out from the heap. Thereafter, by $[e \mapsto e'] \mathrel{-\!\!\star} \ldots$, we extend the heap by a single cell with address e and content e'. After performing the aforementioned two operations, we measure X.

Heap lookup. The statement $x := <e>$ determines the value at address e and stores it in variable x. Its weakest preexpectation is defined as $\sup_{v \in \mathbb{Z}} [e \mapsto v] \star \big([e \mapsto v] \mathrel{-\!\!\star} X[x/v]\big)$. We give an intuition on this preexpectation on a syntactic level. By $[e \mapsto v] \star \ldots$, we ensure that the heap contains a cell with address e and content v, and carve it out from the heap. It is noteworthy that

RIGHTSLINK

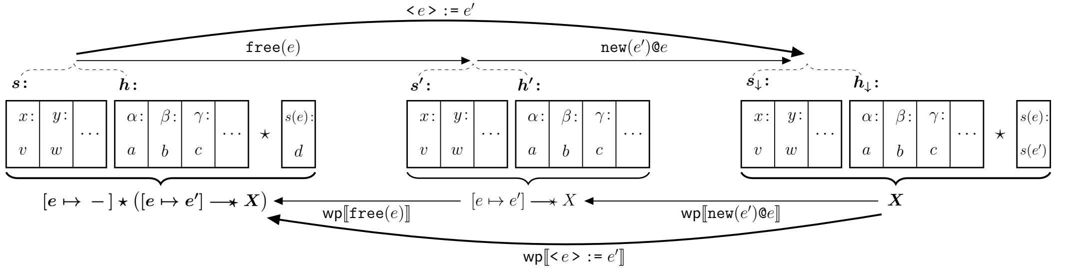

Fig. 6. Weakest preexpectation of heap mutation.

the value v at address e is really *selected* (rather than maximized) by $\sup_{v \in \mathbb{Z}}$. This is because either address e is not allocated at all (i.e. $[e \mapsto v]$ becomes 0 for all choices of v), or there is a *unique* value v at address e which is selected by $\sup_{v \in \mathbb{Z}}$ (i.e. $[e \mapsto v]$ becomes 1). We can thus think of the sup here as taking the role of a $\exists!$-quantifier. After carving out the cell with address e and content v, this very cell is put back into the heap by $[e \mapsto v] \mathrel{-\!\!\star} \ldots$. The aforementioned two operations serve only as a mechanism for selecting v at address e as we can now measure X in a state where variable x has value v through finally measuring $X[x/v]$. Notice that $\sup_{v \in \mathbb{Z}} [e \hookrightarrow v] \cdot X[x/v]$ is equivalent to $\mathsf{wp}[\![x := < e >]\!]\,(X)$ (cf. [Batz et al. 2018]).

4.3 On continuity of wp

For an initially empty heap, the allocation instruction $x := \mathtt{new}\,(e)$ nondeterministically assigns a positive natural number to variable x. It is thus a *countably infinitely branching* nondeterministic assignment. The presence of countably infinite nondeterminism in our semantics has dire consequences: Our wp-calculus is *not* continuous. Consider, for instance, an ω-chain of expectations $X_n = [1 \leq x \leq n]$. Moreover, let $h_\emptyset$ be the empty heap. Then, for an arbitrary stack s,

$$\mathsf{wp}[\![x := \mathtt{new}\,(0)]\!]\,\big(\sup\nolimits_n X_n\big)\,(s, h_\emptyset) \;=\; 1 \neq 0 \;=\; \sup_{n \in \mathbb{N}} \mathsf{wp}[\![x := \mathtt{new}\,(0)]\!]\,(X_n)\,(s, h_\emptyset).$$

Why do we not attempt to find an alternative semantics of $x := \mathtt{new}\,(e)$ that restores continuity? There are two main reasons:

First, [Yang and O'Hearn 2002] argue that nondeterministic allocation in SL is *essential* to enable local reasoning in the presence of address arithmetic. Alternative approaches for allocation, such as always picking the smallest available memory cell, would invalidate the frame rule (cf. Section 4.7).

Second, [Apt and Plotkin 1986] show that it is *impossible* to define a (fully abstract) continuous least fixed point semantics, such as our wp-style calculus, that exhibits countably infinite nondeterministic assignments. Without further restrictions, e.g. limiting ourselves to a finite total amount of available memory, there is thus no hope for a continuous weakest preexpectation transformer.

4.4 Weakest Preexpectations of Loops

As is standard in denotational semantics, the weakest preexpectation of a loop $\mathtt{while}\,(\,b\,)\,\{\,c\,\}$ is characterized as a least fixed point of the loop's unrollings. That is, the weakest preexpectation of program $\mathtt{while}\,(\,b\,)\,\{\,c\,\}$ with respect to postexpectation X is given by the least fixed point of

$$\Phi[\![b, c, X]\!](Y) \;=\; [\neg b] \cdot X + [b] \cdot \mathsf{wp}[\![c]\!]\,(Y) \;\;.$$

Unfortunately, since our wp transformer is *not continuous* in general (see Section 4.3), we cannot rely on Kleene's fixed point theorem. However, due to Theorem 4.2, both wp and $\Phi[\![b, c, X]\!]$ are

RIGHTSLINK

monotone. We may thus resort to a constructive version of the more general fixed point theorem due to Tarski and Knaster (cf. [Cousot and Cousot 1979]) for (countable) ordinals:

Theorem 4.3. *For every loop* $\texttt{while}\,(\,b\,)\,\{\,c\,\}$ *and* $X \in \mathbb{E}$*, there exists an ordinal* α *such that*

$$\mathsf{wp}[\![\texttt{while}\,(\,b\,)\,\{\,c\,\}]\!]\,(X) \;=\; \mathsf{lfp}\; Y\boldsymbol{.}\;\Phi[\![b,c,X]\!](Y) \;=\; \Phi^{\alpha}[\![b,c,X]\!](0)\,.$$

Hence, weakest preexpectations of loops are well-defined. Reasoning about the exact least fixed point of a loop may, however, require transfinite arguments. Fortunately, we have an invariant-based rule for reasoning about *upper bounds* on preexpectations of loops, which is easier to discharge.

Theorem 4.4. *For loop* $\texttt{while}\,(\,b\,)\,\{\,c\,\}$ *and expectations* $X, I \in \mathbb{E}$*, we have*

$$\Phi[\![b,c,X]\!](I) \;\preceq\; I \quad \text{implies} \quad \mathsf{wp}[\![\texttt{while}\,(\,b\,)\,\{\,c\,\}]\!]\,(X) \;\preceq\; I\,.$$

In this case, we call I *an* invariant *with respect to program* $\texttt{while}\,(\,b\,)\,\{\,c\,\}$ *and expectation* X*.*

Proof. By the Tarski and Knaster fixed point theorem, $\mathsf{lfp}\; Y\boldsymbol{.}\;\Phi[\![b,c,X]\!](Y)$ is the smallest pre-fixed point of $\Phi[\![b,c,X]\!]$ (cf. [Cousot and Cousot 1979]). It is thus the smallest I satisfying $\Phi[\![b,c,X]\!](I) \preceq I$. Consequently, by Table 1, $\mathsf{wp}[\![\texttt{while}\,(\,b\,)\,\{\,c\,\}]\!]\,(X) = \mathsf{lfp}\; Y\boldsymbol{.}\;\Phi[\![b,c,X]\!](X) \preceq I$. □

4.5 Soundness of Weakest Preexpectations

We prove the soundness of our weakest preexpectation semantics with respect to the operational semantics introduced in Section 2. To capture the expected value expectation $X \in \mathbb{E}$, we assign a *reward* to every program configuration. Our operational model is a special case of Markov decision process with rewards (cf. [Baier and Katoen 2008; Puterman 2005]). Let $\mathcal{G} \;=\; \{(\Downarrow, s, h) \mid (s,h) \in \Sigma\}$ be the collection of all (goal) configurations indicating successful program termination. Given $X \in \mathbb{E}$, goal configuration $(\Downarrow, s, h)$ is assigned reward $X(s,h)$. All other configurations are assigned zero reward. Formally, the *reward function* for expectation X is given by

$$\mathsf{rew}\colon \mathsf{Conf} \longrightarrow \mathbb{R}_{\geq 0}^{\infty}, \qquad (c,s,h) \mapsto [c = \Downarrow] \cdot X(s,h)\,.$$

We are interested in the minimal (due to demonic nondeterminism) *expected reward* of reaching a goal configuration in $\mathcal{G}$ (and thus successfully terminating) from an initial configuration $\mathsf{init} \in \mathsf{Conf}$. Intuitively, the expected reward is given by the minimal (for all resolutions of nondeterminism) sum over all finite paths π from init to a configuration in $\mathcal{G}$ weighted by the probability of path π and the reward of the reached goal configuration.

Formally, nondeterminism is resolved by a *scheduler* $\rho\colon \mathsf{Conf}^{+} \longrightarrow \mathbb{N}$ mapping finite sequences of visited configurations to the next action. Moreover, let Prob be a function collecting the total probability mass of execution steps ($\longrightarrow$) between two configurations for a given action:

$$\mathsf{Prob}\colon \mathsf{Conf} \times \mathbb{N} \times \mathsf{Conf} \longrightarrow [0,1] \cap \mathbb{Q}, \qquad (t,n,t') \mapsto \sum_{t \xrightarrow{n,p} t'} p\,.$$

The set of *finite paths* from $t \in \mathsf{Conf}$ to some goal configuration using scheduler ρ is given by

$$\begin{aligned}\Pi[t]\,(\rho) \;=\; \{t_1 \ldots t_m \mid{}& m \in \mathbb{N},\; t_1 = t,\; t_m \in \mathcal{G}, \\ & \forall k \in \{1,\ldots,m-1\} \;:\; \mathsf{Prob}(t_k,\; \rho(t_1 \ldots t_k),\; t_{k+1}) > 0\}\,.\end{aligned}$$

The *probability* of a path $t_1 \ldots t_m \in \Pi[t]\,(\rho)$ is the product of its transition probabilities, i.e.

$$\mathsf{Prob}(t_1 \ldots t_m) \;=\; \prod_{1 \leq k < m} \mathsf{Prob}(t_k,\; \rho(t_1 \ldots t_k),\; t_{k+1})\,.$$

RIGHTSLINK

With these notions at hand, the *expected reward of successful termination* with respect to expectation $X \in \mathbb{E}$ when starting execution in configuration $t \in \mathsf{Conf}$ is defined as

$$\mathsf{ExpRew}[\![X]\!]\,(t) \;=\; \inf_{\rho} \sum_{t_1 \ldots t_m \in \Pi[t](\rho)} \mathsf{Prob}(t_1 \ldots t_m) \cdot \mathsf{rew}(t_m)\,.$$

The main result of this subsection asserts that our weakest preexpectation calculus for hpGCL programs is sound with respect to our operational model.

Theorem 4.5 (Soundness of Weakest Preexpectation Semantics). *For all* hpGCL*-programs* c*, expectations* $X \in \mathbb{E}$*, and initial states* $(s, h) \in \Sigma$*, we have* $\mathsf{wp}[\![c]\!]\,(X)\,(s, h) = \mathsf{ExpRew}[\![X]\!]\,(c, s, h)$.

4.6 Conservativity

QSL is a conservative extension of both the weakest preexpectation calculus of [McIver and Morgan 2005] and classical separation logic as developed in [Ishtiaq and O'Hearn 2001; Reynolds 2002]. Since, for programs that never access the heap, we use the same expectation transformer as [McIver and Morgan 2005], it is immediate that QSL conservatively extends weakest preexpectations.

To show that QSL is also a conservative extension of separation logic, recall from Definition 3.3, p. 9, the embedding $\mathtt{qsl}[\![.]\!] \colon \mathsf{SL} \to \mathsf{QSL}$ of SL formulas into QSL. We then obtain conservativity with respect to separation logic in the following sense:

Theorem 4.6 (Conservativity of QSL as a verification system). *Let* $c \in$ hpGCL *be a non-probabilistic program. Then, for all classical separation logic formulas* $\varphi, \psi \in \mathsf{SL}$*,*

$$\textit{the Hoare triple } \{\,\varphi\,\}\, c\, \{\,\psi\,\} \textit{ is valid for total correctness} \quad \textit{iff} \quad \mathtt{qsl}[\![\varphi]\!] \;\preceq\; \mathsf{wp}[\![c]\!]\,\big(\mathtt{qsl}[\![\psi]\!]\big)\,.$$

A key principle underlying separation logic is that correct programs must be memory safe (cf. [Reynolds 2002]), i.e. all executions of a program do not lead to a memory error. By the above theorem, the same holds for our wp calculus when considering non-probabilistic programs. For probabilistic programs, however, we get a more fine-grained view as we can quantify the probability of encountering a memory error. This allows to evaluate programs if failures are unavoidable, for example due to unreliable hardware. In particular, the weakest preexpecation $\mathsf{wp}[\![c]\!]\,(1)$ measures the probability that program c terminates without a memory fault. Does this mean that—for probabilistic programs—our calculus can only prove memory safety with probability one, but is unable to prove that a program is *certainly* memory safe? After all, there might exist an execution of program c that encounters a memory error with probability zero. The answer to this question is *no*: Assume there is some execution of a program c that encounters a memory error. By the correspondence between wp and our operational semantics (cf. Theorem 4.5), there is a path from some initial state to an error state $(\lightning, s, h)$. Since such a path must be finite and thus has a positive probability, the probability of encountering a memory error must be positive. In other words,

Corollary 4.7. *An* hpGCL *program is memory safe with probability one iff it is memory safe.*

4.7 The Quantitative Frame Rule

In classical SL (in the sense of a proof system), the *frame rule* is a distinguished feature that allows for local reasoning [Yang and O'Hearn 2002]. Intuitively, it states that a part of the heap that is not explicitly modified by a program is unaffected by that program. Consequently, it suffices to reason locally only on the subheap that is actually mutated. The frame rule reads as follows:

$$\frac{\langle\,\varphi\,\rangle\; c\; \langle\,\psi\,\rangle}{\langle\,\varphi \star \vartheta\,\rangle\; c\; \langle\,\psi \star \vartheta\,\rangle} \quad \text{if } \mathit{Mod}(c) \cap \mathsf{Vars}(\vartheta) = \emptyset.$$

RIGHTSLINK

Here, $Mod(c)$ is the set of variables updated by a program c, i.e. all variables appearing on a left-hand side of an assignment in c.[6] Moreover, $\mathsf{Vars}(\vartheta)$ collects all variables that "occur" in ϑ.[7]

Towards a quantitative frame rule. Let us first translate the above Hoare-style rule into an equivalent version for weakest preconditions. To this end, we use the well-established fact that

$$\langle \varphi \rangle\ c\ \langle \psi \rangle \text{ is valid} \quad \text{iff} \quad \varphi \Rightarrow \mathsf{wp}[\![c]\!]\,(\psi)\ .$$

Notice that this fact remains valid for memory-fault avoiding interpretations of Hoare triples as used by [Yang and O'Hearn 2002]. Based on this fact, we obtain a suitable formulation of the frame rule in the setting of weakest preconditions: Assume that $Mod(c) \cap \mathsf{Vars}(\vartheta) = \emptyset$. Then

$$\frac{\langle \varphi \rangle\ c\ \langle \psi \rangle}{\langle \varphi \star \vartheta \rangle\ c\ \langle \psi \star \vartheta \rangle}$$

$$\text{iff} \qquad (\varphi \Rightarrow \mathsf{wp}[\![c]\!]\,(\psi)) \quad \Rightarrow \quad (\varphi \star \vartheta \Rightarrow \mathsf{wp}[\![c]\!]\,(\psi \star \vartheta)) \qquad (\clubsuit)$$

$$\text{iff} \qquad \mathsf{wp}[\![c]\!]\,(\psi) \star \vartheta \quad \Rightarrow \quad \mathsf{wp}[\![c]\!]\,(\psi \star \vartheta)\ . \qquad (\spadesuit)$$

To understand the last equivalence, assume that ($\clubsuit$) holds and choose $\varphi = \mathsf{wp}[\![c]\!]\,(\psi)$. Then replacing φ by $\mathsf{wp}[\![c]\!]\,(\psi)$ in the conclusion of ($\clubsuit$) immediately yields the implication ($\spadesuit$). Conversely, assume ($\spadesuit$) holds and let $\varphi \Rightarrow \mathsf{wp}[\![c]\!]\,(\psi)$. By monotonicity of $\star$, we obtain $\varphi \star \vartheta \Rightarrow \mathsf{wp}[\![c]\!]\,(\psi) \star \vartheta$. Then ($\spadesuit$) yields that $\mathsf{wp}[\![c]\!]\,(\psi) \star \vartheta$ implies $\mathsf{wp}[\![c]\!]\,(\psi \star \vartheta)$, i.e. ($\clubsuit$) holds.

In a quantitative setting the analog to implication $\Rightarrow$ is $\preceq$. Hence, the frame rule for QSL is:

Theorem 4.8 (Quantitative Frame Rule). *For every* hpGCL*-program c and expectations $X, Y \in \mathbb{E}$ with $Mod(c) \cap \mathsf{Vars}(Y) = \emptyset$, we have* $\mathsf{wp}[\![c]\!]\,(X) \star Y \preceq \mathsf{wp}[\![c]\!]\,(X \star Y)$.

Proof. By structural induction on hpGCL programs. For loops, we additionally have to perform a transfinite induction on the number of iterations. □

What about the converse direction? Can we also obtain a frame rule of the form $\mathsf{wp}[\![c]\!]\,(X) \star Y \succeq \mathsf{wp}[\![c]\!]\,(X \star Y)$? In the quantitative case, a converse frame rule breaks for probabilistic choice due to the fact that $\star$ and $+$ are only subdistributive in general (Theorem 3.6). This problem can partially be avoided by requiring Y to be domain-exact. However, the "converse frame rule" also breaks in the qualitative case, i.e. if X and Y are predicates and $\succeq$ corresponds to $\Leftarrow$: For $X = [\mathbf{emp}]$, we have

$$\mathsf{wp}[\![<x> := 0]\!]\,([\mathbf{emp}]) \;=\; [x \mapsto -] \star ([x \mapsto 0] \mathrel{-\!\!\star} [\mathbf{emp}]) \;=\; 0\ .$$

If we additionally choose $Y = [x \hookrightarrow 0]$, we also obtain $Mod(c) \cap \mathsf{Vars}(Y) = \emptyset$ and

$$\begin{aligned} \mathsf{wp}[\![<x> := 0]\!]\,([\mathbf{emp}] \star [x \hookrightarrow 0]) \;&=\; [x \mapsto -] \star ([x \mapsto 0] \mathrel{-\!\!\star} ([\mathbf{emp}] \star [x \hookrightarrow 0])) \\ &=\; [x \hookrightarrow -]\ . \end{aligned}$$

Put together, this yields a counterexample—even in the *qualitative* case:

$$\mathsf{wp}[\![<x> := 0]\!]\,([\mathbf{emp}]) \star [x \hookrightarrow 0] \;\not\succeq\; \mathsf{wp}[\![<x> := 0]\!]\,([\mathbf{emp}] \star [x \hookrightarrow 0])\ .$$

Hence, there is no converse version of the frame rule for a conservative extension of SL.

5 A LANDSCAPE OF WEAKEST PREEXPECTATION CALCULI

Our weakest preexpectation calculus for QSL is for total correctness with intrinsic memory safety and demonic nondeterminism. We now briefly discuss alternative possibilities.

[6] More formally, $Mod(c) = \{x\}$ if c is of the form $x := e$, $x := \mathtt{new}(\vec{e})$, or $x := <e>$, and $Mod(c) = \emptyset$ if c is $\mathtt{skip}$, $\mathtt{free}(e)$, or $<e> := e'$. For the composed programs, we have $Mod(c) = Mod(c_1) \cup Mod(c_2)$ if c is either $\mathtt{if}\ (b)\ \{c_1\}\ \mathtt{else}\ \{c_2\}$, $\{c_1\}\ [p]\ \{c_2\}$, or $c_1 ; c_2$. For loops, we have $Mod(\mathtt{while}\,(b)\,\{c\}) = Mod(c)$.

[7] Formally, $x \in \mathsf{Vars}(\vartheta)$ iff $\exists (s, h) \in \Sigma\ \exists v, v' \in \mathbb{Z}\colon \vartheta(s[x/v], h) \neq \vartheta(s[x/v'], h)$.

RIGHTSLINK

Angelic nondeterminism. For a program c and $X \in \mathbb{E}$, instead of the *least* expected value $\mathsf{wp}[\![c]\!]\,(X)$, we are now interested in the *largest* expected value $\mathsf{awp}[\![c]\!]\,(X)$ (read: angelic weakest preexpectation) of X after execution of c. How does angelic nondeterminism affect the inductive definition of wp in Table 1? For nondeterministic statements, we now have to maximize instead of minimize the expected value. As $x :\approx \mathtt{new}\,(\vec{e})$ is the only statement that exhibits nondeterminism, we get

$$\mathsf{awp}[\![x :\approx \mathtt{new}\,(\vec{e})]\!]\,(X) \;=\; \lambda(s,h)\textbf{.} \sup_{v \in \mathbb{N}_{>0}: v, v+1, \ldots, v+|\vec{e}|-1 \notin \mathsf{dom}(h)} ([v \mapsto \vec{e}] \mathrel{-\!\!\star} X\,[x/v])\,(s,h)\,,$$

where, since allocation never fails, we only choose from locations that are not already allocated. For all other statements, awp is defined just as wp in Table 1 (except that wp is replaced by awp).

Since a question like "what is the expected value of x after execution of program c" does not make much sense if there is some positive probability such that c does not terminate or encounters a memory fault, the remainder of this section considers expectations in $\mathbb{E}_{\leq 1}$ only.

Partial correctness. The weakest *liberal* preexpectation $\mathsf{wlp}[\![c]\!]\,(X)\,(s,\,h)$ of program c and expectation $X \in \mathbb{E}_{\leq 1}$ for an initial state $(s,\,h)$ corresponds to the weakest preexpectation $\mathsf{wp}[\![c]\!]\,(X)\,(s,\,h)$ plus the probability that c does not terminate on state $(s,\,h)$. How does shifting to partial correctness affect the inductive definition of wp in Table 1? Following [McIver and Morgan 2005], we consider the *greatest* fixed point for loops:

$$\mathsf{wlp}[\![\mathtt{while}\,(\,b\,)\,\{\,c'\,\}]\!]\,(X) \;=\; \mathsf{gfp}\;Y\textbf{.}\;\underbrace{[\neg b] \cdot X + [b] \cdot \mathsf{wlp}[\![c']\!]\,(Y)}_{=\,\Phi[\![b,c,X]\!](Y)}\;.$$

For weakest liberal preexpectations, our quantitative frame rule also applies:

Theorem 5.1 (Quantitative Frame Rule for wlp). *For every* hpGCL*-program c and expectations $X, Y \in \mathbb{E}_{\leq 1}$ with $Mod(c) \cap \mathsf{Vars}(Y) = \emptyset$, we have* $\mathsf{wlp}[\![c]\!]\,(X) \star Y \preceq \mathsf{wlp}[\![c]\!]\,(X \star Y)$.

Furthermore, a dual version of our proof rule for invariant-based reasoning about loops is available for weakest liberal preexpectations. Its proof is analogous to the proof of Theorem 4.4.

Theorem 5.2. *For loop* $\mathtt{while}\,(\,b\,)\,\{\,c\,\}$*, postexpectation $X \in \mathbb{E}_{\leq 1}$ and invariant $I \in \mathbb{E}_{\leq 1}$, we have*

$$I \;\preceq\; \Phi[\![b,c,X]\!](I) \quad \text{implies} \quad I \;\preceq\; \mathsf{wlp}[\![\mathtt{while}\,(\,b\,)\,\{\,c\,\}]\!]\,(X)\;.$$

Extrinsic memory safety. Finally, we assume terminating with a memory fault is acceptable. This is analogous to weakest liberal preexpectations, where nontermination is considered acceptable. The weakest *extrinsic memory safe* preexpectation $\mathsf{wep}[\![c]\!]\,(X)\,(s,\,h)$ corresponds to the weakest preexpectation $\mathsf{wp}[\![c]\!]\,(X)$ plus the probability that c terminates with a memory fault on initial state $(s,\,h)$. How does extrinsic memory safety affect the inductive definition of wp in Table 1? We have to modify the connectives $\star$ and $\mathrel{-\!\!\star}$ to add the probability of memory faults. The resulting connectives, denoted $X \bullet Y$ and $[\varphi] \mathrel{-\!\!\bullet} Y$, where $X, Y \in \mathbb{E}_{\leq 1}$ and φ is a predicate, are defined below.

$$\begin{aligned} X \bullet Y \;&=\; \lambda(s,h)\textbf{.}\;\min\,\{\, 1 - X(s,h_1) + X(s,h_1) \cdot Y(s,h_2) \mid h = h_1 \star h_2 \,\} \\ [\varphi] \mathrel{-\!\!\bullet} Y \;&=\; \lambda(s,h)\textbf{.}\;\sup_{h'}\,\{Y(s,h \star h') \mid h \perp h' \text{ and } (s,h') \models \varphi\} \end{aligned}$$

The rules of wep are then obtained from the rules for wp in Table 1 by replacing every occurrence of $\star$ by $\bullet$ and $\mathrel{-\!\!\star}$ by $\mathrel{-\!\!\bullet}$, respectively, and changing the rule for heap lookups to

$$\mathsf{wep}[\![x := \mathtt{<}\,e\,\mathtt{>}]\!]\,(X) \;=\; \inf_{v \in \mathbb{Z}}\,[e \mapsto v] \bullet ([e \mapsto v] \mathrel{-\!\!\bullet} X\,[x/v])\;.$$

Thus, we replaced the supremum by an infimum as encountering a memory fault is acceptable.

RIGHTSLINK

The weakest preexpectation landscape. The individual changes to wp can easily be combined. Thus, apart from wp, awp, wlp, and wep, we also have transformers awlep, awep, wlep, and awlp. How are these transformers related? As a first observation, we note that for every hpGCL-program c, we have $\mathsf{wp}[\![c]\!]\,(0) = 0$ and $\mathsf{awlep}[\![c]\!]\,(1) = 1$. For each given initial state $(s,\, h)$ there are at most four possible outcomes: c *diverges*, c encounters a memory *fault*, c successfully terminates in a state "captured by X", or c terminates in some other state, which we denote by $\neg X$. The total probability of these four outcomes is one. Hence, we can describe the probability of successful termination and measuring X, i.e. $\mathsf{wp}[\![c]\!]\,(X)$, as one minus the probability of the other three events. Similar dualities are obtained for all of the possible calculi:

Theorem 5.3 (Duality principle for the weakest preexpectation landscape). *Let $c \in$ hpGCL be a program. Moreover, let $X \in \mathbb{E}_{\leq 1}$. Then*

$$\begin{aligned}
\mathsf{wp}[\![c]\!]\,(X) &= 1 - \mathsf{awlep}[\![c]\!]\,(1-X)\;, && \text{(probability of } X\text{)}\\
\mathsf{wlp}[\![c]\!]\,(X) &= 1 - \mathsf{awep}[\![c]\!]\,(1-X)\;, && \text{(probability of } X + \text{divergence)}\\
\mathsf{wep}[\![c]\!]\,(X) &= 1 - \mathsf{awlp}[\![c]\!]\,(1-X)\;,\ \textit{and} && \text{(probability of } X + \text{memory fault)}\\
\mathsf{wlep}[\![c]\!]\,(X) &= 1 - \mathsf{awp}[\![c]\!]\,(1-X)\;. && \text{(probability of } X + \text{divergence} + \text{memory fault)}
\end{aligned}$$

6 BEYOND HPGCL PROGRAMS

We presented our results in terms a simple probabilistic programming language. Some of the case studies presented in the next section, however, additionally use procedure calls and sample from discrete uniform probability distributions. Let us thus briefly discuss how our wp calculus is extended accordingly.[8]

We allow programs c to contain *procedure calls* of the form $\mathtt{call}\ \mathsf{P}\,(\vec{e})$, where P is a procedure name and $\vec{e}$ is a tuple of arithmetic expressions representing the values passed to the procedure. Since assume parameters are passed by value, no variables are modified by a procedure call, i.e. $\mathit{Mod}(\mathtt{call}\ \mathsf{P}\,(\vec{e})) = \emptyset$. The meaning of procedure calls is determined by *procedure declarations* of the form $\mathtt{procedure}\ \mathsf{P}\,(\vec{x})\,\{\, \mathit{body}(P)\,\}$, where $\mathit{body}(P) \in$ hpGCL is the procedure's body that may contain (recursive) procedure calls and $\vec{x}$ is a tuple of variables that are never changed by program $\mathit{body}(P)$. All variables in $\mathit{body}(P)$ except for its parameters are considered local variables.

For non-recursive procedures, the weakest preexpectation of a procedure call coincides with the weakest preexpectation of its body. The semantics of recursive procedure calls is determined by a least fixed point of a transformer on procedure environments mapping procedure names and parameters to expectations. In particular, our previous results, such as linearity of wp, monotonicity, and the frame rule, remain valid in the presence of recursive procedure calls.

Furthermore, we employ a standard proof rule to deal with recursion (cf. [Hesselink 1993]):

$$\frac{\forall \vec{e}\colon \mathsf{wp}[\![\mathtt{call}\ \mathsf{P}\,(\vec{e})]\!]\,(X) \;\preceq\; I(\vec{e}) \;\Vdash\; \mathsf{wp}[\![\mathit{body}(\mathsf{P})]\!]\,(X) \;\preceq\; I(\vec{e})}{\forall \vec{e}\colon \mathsf{wp}[\![\mathtt{call}\ \mathsf{P}\,(\vec{e})]\!]\,(X) \;\preceq\; I(\vec{e}),}\ [\text{rec}]$$

where $X \in \mathbb{E}$ is a postexpectation, $I(\vec{e}) \in \mathbb{E}$ is an invariant, and $\vec{e}$ is a tuple of expression passed to the called procedure. Intuitively, for proving that a procedure call satisfies a specification, it suffices to show that the procedures body satisfies the specification—assuming that all recursive calls in the procedure's body do so, too. An analogous rule is obtained for weakest *liberal* preexpectations by replacing all occurrences of $\preceq$ by $\succeq$.

Moreover, we support sampling from arbitrary discrete distributions instead of flipping coins. While these sampling instructions, such as $x \mathrel{:=} \mathtt{uniform}\,(e, e')$, which samples an integer in the interval $[e, e']$ uniformly at random, can be simulated with coin flips, it is more convenient to

[8] Detailed formalizations and extensions of previous proofs are found in [Batz et al. 2018].

RIGHTSLINK

directly derive their semantics. For example, the weakest preexpecation of the *uniform random assignment* x := uniform(e, e') with respect to postexpectation $X \in \mathbb{E}$ is given by

$$\mathsf{wp}[\![x := \mathtt{uniform}(e, e')]\!](X) \;=\; \lambda(s,h)\textbf{.}\; \frac{1}{s(e') - s(e) + 1} \cdot \sum_{k=s(e)}^{s(e')} X[x/k](s,h)\,.$$

7 CASE STUDIES

We examine a few examples—including the programs presented in Section 1—to demonstrate QSL's applicability to reason about probabilities and expected values of hpGCL programs.

7.1 Array Randomization

For our first example, recall the procedure randomize(array, n) in Section 1, Figure 1a that computes a random permutation of an array of size n. To conveniently specify subarrays, we use iterated separating conjunctions (cf. [Reynolds 2002]) given by

$$\bigstar_{k=i}^{n} X_k \;=\; \lambda(s,h)\textbf{.} \begin{cases} \left(X_{s(i)} \star X_{s(i+1)} \star \ldots \star X_{s(n)}\right)(s,h) & \text{if } s(i) \leq s(n) \\ [\mathbf{emp}](s,h) & \text{otherwise.} \end{cases}$$

Our goal is to show that no particular permutation of the input array has a higher probability than other ones. Since there are $n!$ permutations of an array of length n, we prove that the probability of computing an arbitrary, but fixed, permutation is at most $1/n!$. That is, we compute an upper bound of $\mathsf{wp}[\![\mathtt{call\ randomize}(\mathtt{array}, n)]\!]([\mathtt{array} \mapsto \alpha_0, \ldots, \alpha_{n-1}])$, where we use variables $\alpha_0, \ldots, \alpha_{n-1}$, which do not appear in the program, to keep track of the individual values in the array. To this end, we propose the invariant

$$\begin{aligned} I \;=\; & [0 \leq i < n] \cdot \frac{1}{(n-i)!} \cdot \bigstar_{k=0}^{i-1} [\mathtt{array} + k \mapsto \alpha_k] \star \sum_{\pi \in \mathrm{Perm}(i, n-1)} \bigstar_{k=i}^{n-1} \left[\mathtt{array} + k \mapsto \alpha_{\pi(k)}\right] \\ & + [\neg(0 \leq i < n)] \cdot [\mathtt{array} \mapsto \alpha_0, \ldots, \alpha_{n-1}] \end{aligned}$$

for the loop c_{loop} in procedure randomize, where $\mathrm{Perm}(e, e')$ denotes the set of permutations over $\{e, e+1, \ldots, e'\}$. Intuitively, I describes the situation for i *remaining* loop iterations (since we reason backwards): All but the first i array elements are already known to be swapped consistently with our fixed permutation. In our preexpectation, the last $n - i$ elements are thus arbitrarily permuted and the probability of hitting the right permutation for these elements is $1/(n-i)!$. The remaining i iterations still have to be executed, i.e. the first i array elements coincide with our postexpectation. For the whole procedure randomize we continue as follows:

$$\begin{aligned} & \mathsf{wp}[\![\mathtt{call\ randomize}(\mathtt{array}, n)]\!]([\mathtt{array} \mapsto \alpha_0, \ldots, \alpha_{n-1}]) \\ =\; & \mathsf{wp}[\![i := 0]\!]\left(\mathsf{wp}[\![c_{\text{loop}}]\!]([\mathtt{array} \mapsto \alpha_0, \ldots, \alpha_{n-1}])\right) && \text{(Definition of wp for procedure body)} \\ \preceq\; & \mathsf{wp}[\![i := 0]\!](I) \;=\; I[i/0] && \text{(Theorem 4.4 for invariant } I\text{, Table 1)} \\ =\; & \frac{1}{n!} \cdot \sum_{\pi \in \mathrm{Perm}(0, n-1)} \bigstar_{k=0}^{n-1} \left[\mathtt{array} + k \mapsto \alpha_{\pi(k)}\right]. && \text{(Algebra)} \end{aligned}$$

The probability of computing exactly the permutation $\alpha_0, \ldots, \alpha_{n-1}$ is thus at most $1/n!$. Moreover, if the initial heap is not some permutation of our fixed array, the probability becomes 0.

7.2 Faulty Garbage Collector

The next example is a garbage collector that is executed on cheap, but unreliable hardware (cf. Section 1): Procedure delete takes a binary tree with root x and recursively deletes all elements in

RIGHTSLINK

```
procedure delete (x) { // [tree (x)] · (1 − p)^size                                          (7)
  // [x ≠ 0] · (p · [emp] + (1 − p) · f) + [x = 0] · [emp]                                   (6)
  if ( x ≠ 0 ) { // p · [emp] + (1 − p) · f                                                  (5)
    { skip } [p] { // sup_{α,β} [x ↦ α, β] ⋆ ([x ↦ α, β] −⋆ g [l, r/α, β]) =: f              (4)
      l := < x >; r := < x + 1 >; // [x ↦ −, −] ⋆ t(l) ⋆ t(r) =: g                          (3)
      call delete (l) ; // [x ↦ −, −] ⋆ t(r)                                                 (2)
      call delete (r) ; // [x ↦ −, −] ⋆ [emp]                                                (1)
      free(x); free(x + 1) } // [emp]
  } else {skip} } // [emp]
} // [emp]
```

Fig. 7. Faulty garbage collection procedure with a proof sketch, where $t(\alpha) \;=\; [\mathrm{tree}\,(\alpha)] \cdot (1-p)^{\mathbf{size}}$.

the tree. However, with some probability $p \in [0, 1]$, the condition $x \neq 0$, which checks whether the tree is empty, is ignored although x is the root of a non-empty tree. This scenario is implemented by the probabilistic program in Figure 7, where each node in a tree consists of two consecutive pointers: $<\alpha>$ and $<\alpha+1>$ respectively represent the left and right child of α.

Our goal is to establish a lower bound on the probability that the garbage collector successfully deletes the whole tree, i.e. $\mathsf{wlp}[\![\texttt{call delete}\,(x)]\!]\,([\mathbf{emp}])$. To this end we claim that

$$\mathsf{wlp}[\![\texttt{call delete}\,(x)]\!]\,([\mathbf{emp}]) \;\;\geq\;\; [\mathrm{tree}\,(x)] \cdot (1-p)^{\mathbf{size}} \;, \qquad (\dagger)$$

where $\left(p^X\right)(s,\,h) = p^{X(s,h)}$ for some rational p and $X \in \mathbb{E}_{\leq 1}$. The main steps of a proof of our claim are sketched in Figure 7: Starting with postexpectation $[\mathbf{emp}]$, step (1) results from applying the wlp rule for $\texttt{free}(_)$ and the fact that

$$[x \mapsto -] \star [x+1 \mapsto -] \star [\mathbf{emp}] \;\;=\;\; [x \mapsto -, -] \star [\mathbf{emp}] \;.$$

Step (2) deserves special attention. We would like to apply rule $[rec]$ for recursive procedures (and wlp) using the premise

$$\mathsf{wlp}[\![\texttt{call delete}\,(r)]\!]\,([\mathbf{emp}]) \;\;\geq\;\; t(r) \;\;=\;\; [\mathrm{tree}\,(r)] \cdot (1-p)^{\mathbf{size}},$$

but the postexpectation is $[x \mapsto -, -] \star [\mathbf{emp}]$ instead of $[\mathbf{emp}]$. Here, the quantitative frame rule (Theorem 4.8) allows us to apply the rule $[rec]$ for recursive procedures to postexpectation $[\mathbf{emp}]$ and derive $[x \mapsto -, -] \star t(r)$. Notice that the frame rule would not be applicable without the separating conjunction. In particular, our proof would have to deal with *aliasing*: It is not immediate that the heaps reachable from l and r do not share memory.

Step (3) first extends the postexpectation exploiting that $Z = Z \star [\mathbf{emp}]$ for any $Z \in \mathbb{E}_{\leq 1}$. We then proceed analogously to step (2). Step (4) is an application of the lookup rule with minor simplifications to improve readability. Steps (5) and (6) apply wlp to the probabilistic choice and the conditional. Finally, we show that (6) is entailed by the expectation in step (7), i.e. (6) $\geq$ (7).

7.3 Lossy List Reversal

We analyze the lossy list reversal presented in Section 1, Figure 1b. Our goal is to obtain an upper bound on the *expected length of the reversed list* after successful termination, i.e. we compute an

RIGHTSLINK

upper bound of $\mathsf{wp}[\![\mathtt{call\ lossyReversal}\,(hd)]\!]\,(\mathsf{len}\,(r,0))$. To this end, we propose the invariant

$$I \;=\; \mathsf{len}\,(r,0) \star [\mathsf{ls}\,(hd,0)] + 1/2 \cdot [hd \neq 0] \cdot (\mathsf{len}\,(hd,0) \star [\mathsf{ls}\,(r,0)])\,.$$

Intuitively, invariant I states that during each loop iteration, the expected length of the list with head r is its current length, i.e. $\mathsf{len}\,(r,0)$, plus half of the length of the remaining list with head hd, i.e. $\mathsf{len}\,(hd,0)$. To obtain a tight specification, i.e. describe the exact content of the heap, we additionally use predicates $[\mathsf{ls}\,(hd,0)]$ and $[\mathsf{ls}\,(r,0)]$ to cover the remaining parts of the heap when measuring the length of a list. We then continue as follows:

$$\begin{aligned}
& \mathsf{wp}[\![\mathtt{call\ lossyReversal}\,(hd)]\!]\,(\mathsf{len}\,(r,0)) \\
=\;& \mathsf{wp}[\![r := 0]\!]\,\big(\mathsf{wp}[\![\mathtt{while}\,(\,hd \neq 0\,)\,\{\ldots\}]\!]\,(\mathsf{len}\,(r,0))\big) && \text{(Definition of procedure body, Table 1)} \\
\preceq\;& \mathsf{wp}[\![r := 0]\!]\,(I) && \text{(Theorem 4.4)} \\
=\;& \underbrace{\mathsf{len}\,(0,0)}_{=0} \star [\mathsf{ls}\,(hd,0)] + 1/2 \cdot [hd \neq 0] \cdot (\mathsf{len}\,(hd,0) \star \underbrace{[\mathsf{ls}\,(0,0)]}_{=1}) && \text{(Def. of } I\text{, Table 1)} \\
=\;& 1/2 \cdot [hd \neq 0] \cdot \mathsf{len}\,(hd,0)\,.
\end{aligned}$$

Hence, the expected length of the reversed list after successful termination is at most half of the length of the original list.

7.4 Randomized List Extension

As a last example, we consider a program c_{list} that inserts new elements at the beginning of a list with head x, but gradually loses interest in adding further elements:

$$c_{\text{list}}: \quad c := 1\,;\ \mathtt{while}\,(\,c = 1\,)\,\{\,\{\,c := 0\,\}\,[\,1/2\,]\,\{\,c := 1\,;\ x := \mathtt{new}\,(x)\,\}\,\}$$

Our goal is to compute an upper bound on the expected length of the list with head x after termination of program c_{list}, i.e. we compute an upper bound of $\mathsf{wp}[\![c_{\text{list}}]\!]\,(\mathsf{len}\,(x,0))$. To this end, we propose the loop invariant $I = \mathsf{len}\,(x,0) + [c = 1]$, which states that the length of the list is increased by one if variable c equals one. For the full program we proceed as follows:

$$\begin{aligned}
& \mathsf{wp}[\![c_{\text{list}}]\!]\,(\mathsf{len}\,(x,0)) \\
=\;& \mathsf{wp}[\![c := 1]\!]\,\big(\mathsf{wp}[\![\mathtt{while}\,(\,c = 1\,)\,\{\ldots\}]\!]\,(\mathsf{len}\,(x,0))\big) && \text{(Definition of wp)} \\
\preceq\;& \mathsf{wp}[\![c := 1]\!]\,(I) \;=\; I\,[c/1] && \text{(Theorem 4.4 for invariant } I\text{)} \\
=\;& \mathsf{len}\,(x,0) + 1. && \text{(Algebra)}
\end{aligned}$$

Hence, in expectation, program c_{lists} increases the length of the initial list by at most one element.

8 RELATED WORK

Although many algorithms rely on randomized data structures, formal reasoning about probabilistic programs that mutate memory has received scarce attention. To the best of our knowledge, there is little other work on formal verification of programs that are both probabilistic and heap manipulating. A notable exception is recent work by [Tassarotti and Harper 2018] who combine concurrent separation logic with probabilistic relational Hoare logic (cf. [Barthe et al. 2012]). Their focus is on program refinement. Verification is thus understood as establishing a relation between a program to be analyzed and a program which is known to be well-behaved. In contrast to that, the goal of our logic is to directly measure quantitative program properties on source code level using a weakest-precondition style calculus. In particular, programs that do not *certainly* terminate, e.g. the list extension example in Section 7.4, are outside the scope of their approach (cf. [Tassarotti and Harper 2018, Theorem 3.1]). Furthermore, they do not consider unbounded expectations.

RIGHTSLINK

Probabilistic program verification. Seminal work on semantics and verification of probabilistic programs is due to [Kozen 1979, 1983]. [McIver and Morgan 2005; Morgan et al. 1996] developed the weakest preexpectation calculus to reason about a probabilistic variant of Dijkstra's guarded command language. While variants of their calculus have been successfully applied to programs that access data structures, such as the coupon collector's problem [Kaminski et al. 2016] and a probabilistic binary search [Olmedo et al. 2016], treatment of data structures is usually added in an ad–hoc manner. In particular, proofs quickly get extremely complicated if programs do not only access but also mutate a data structure. Our work extends the calculus of McIver and Morgan to formally reason about heap manipulating probabilistic programs.

Separation Logic. Apart from the backward reasoning rules in [Ishtiaq and O'Hearn 2001; Reynolds 2002], weakest preconditions are extensively used by [Krebbers et al. 2017]. For ordinary programs, our calculus allows for reasoning about quantities of heaps, such as the length of lists. Such shape–numeric properties have been investigated before, see, e.g., [Bozga et al. 2010; Chang and Rival 2008]. [Chin et al. 2012] use recursive predicate definitions together with fold/unfold reasoning to verify properties, such as balancedness of trees. Furthermore, [Atkey 2011] developed a proof logic that combines separation logic with reasoning about consumable resources. His work supports reasoning about quantities by means of special predicates that that are evaluated by one or more resources in addition to the heap. However, the amount of resources must be bounded. It is unclear how this approach can be extended to reason about expected values of probabilistic programs.

9 CONCLUSION

We presented QSL — a quantitative separation logic that evaluates to real numbers instead of truth values. Our wp calculus built on top of QSL is a conservative extension of both separation logic and Kozen's / McIver and Morgan's weakest preexpectations. In particular, virtually all properties of separation logic remain valid. We applied QSL to reason about four examples, ranging from the success probability of a faulty garbage collector, over the expected list length of a list reversal algorithm to a textbook procedure to randomize arrays.

Our calculus provides a foundation for formal reasoning about randomized algorithms on source code level. Future work includes developing *proof systems for quantitative entailments* and analyzing more involved algorithms, e.g. randomized skip lists or randomized splay trees.

ACKNOWLEDGMENTS

We are grateful for the valuable and very constructive comments we received from the anonymous reviewers. This applies particularly to the formulation of Theorems 3.4 and 4.6.

Furthermore, we acknowledge the support of this work by DFG research training group 2236 UnRAVeL and by DFG grant NO 401/2-1.

REFERENCES

Susanne Albers and Marek Karpinski. 2002. Randomized splay trees: Theoretical and experimental results. *Inf. Process. Lett.* 81, 4 (2002), 213–221.

Krzysztof R Apt and Gordon D Plotkin. 1986. Countable nondeterminism and random assignment. *Journal of the ACM (JACM)* 33, 4 (1986), 724–767.

Cecilia R. Aragon and Raimund Seidel. 1989. Randomized Search Trees. In *FOCS*. 540–545.

Robert Atkey. 2011. Amortised Resource Analysis with Separation Logic. *Logical Methods in Computer Science* 7, 2 (2011). https://doi.org/10.2168/LMCS-7(2:17)2011

Christel Baier and Joost-Pieter Katoen. 2008. *Principles of Model Checking*. MIT Press.

Gilles Barthe, Thomas Espitau, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2018. An Assertion-Based Program Logic for Probabilistic Programs. In *ESOP 2018*. 117–144. https://doi.org/10.1007/978-3-319-89884-1_5

RIGHTSLINK

Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. 2012. Probabilistic Relational Hoare Logics for Computer-Aided Security Proofs. In *MPC*. 1–6.
Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Thomas Noll. 2018. Quantitative Separation Logic. *CoRR* abs/1802.10467 (2018). arXiv:1802.10467 http://arxiv.org/abs/1802.10467
Guy E. Blelloch and Margaret Reid-Miller. 1998. Fast Set Operations Using Treaps. In *SPAA*. 16–26.
Marius Bozga, Radu Iosif, and Swann Perarnau. 2010. Quantitative Separation Logic and Programs with Lists. *J. Autom. Reasoning* 45, 2 (2010), 131–156.
James Brotherston. 2007. Formalised Inductive Reasoning in the Logic of Bunched Implications. In *SAS*. 87–103.
Michael Carbin, Sasa Misailovic, and Martin C. Rinard. 2016. Verifying quantitative reliability for programs that execute on unreliable hardware. *Commun. ACM* 59, 8 (2016), 83–91. https://doi.org/10.1145/2958738
Aleksandar Chakarov and Sriram Sankaranarayanan. 2013. Probabilistic Program Analysis with Martingales. In *CAV (LNCS)*, Vol. 8044. Springer, 511–526.
Bor-Yuh Evan Chang and Xavier Rival. 2008. Relational inductive shape analysis. In *POPL*. 247–260.
Krishnendu Chatterjee, Hongfei Fu, Petr Novotný, and Rouzbeh Hasheminezhad. 2016. Algorithmic Analysis of Qualitative and Quantitative Termination Problems for Affine Probabilistic Programs. In *POPL*. ACM, 327–342.
Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. 2012. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Sci. Comput. Program.* 77, 9 (2012), 1006–1036.
Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, 3rd Edition.* MIT Press. http://mitpress.mit.edu/books/introduction-algorithms
Patrick Cousot and Radhia Cousot. 1979. Constructive versions of Tarski's fixed point theorems. *Pacific J. Math.* 82, 1 (1979), 43–57.
Edsger Wybe Dijkstra. 1976. *A Discipline of Programming.* Prentice–Hall.
Rusins Freivalds. 1977. Probabilistic Machines Can Use Less Running Time. In *IFIP Congress*, Vol. 839. 842.
Friedrich Gretz, Joost-Pieter Katoen, and Annabelle McIver. 2014. Operational versus Weakest Pre-Expectation Semantics for the Probabilistic Guarded Command Language. *Performance Evaluation* 73 (2014), 110–132.
Thomas A. Henzinger. 2013. Quantitative reactive modeling and verification. *Computer Science - R&D* 28, 4 (2013), 331–344.
Wim H. Hesselink. 1993. Proof Rules for Recursive Procedures. *Formal Asp. Comput.* 5, 6 (1993), 554–570.
Charles Antony Richard Hoare. 1962. Quicksort. *Comput. J.* 5, 1 (1962), 10–15.
Charles Antony Richard Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (1969), 576–580.
Samin S. Ishtiaq and Peter W. O'Hearn. 2001. BI as an Assertion Language for Mutable Data Structures. In *POPL*. 14–26.
Claire Jones. 1990. *Probabilistic Non–Determinism.* Ph.D. Dissertation. University of Edinburgh, UK.
Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. 2016. Weakest Precondition Reasoning for Expected Run–Times of Probabilistic Programs. In *ESOP (LNCS)*, Vol. 9632. Springer, 364–389.
Donald Ervin Knuth. 1992. Two Notes on Notation. *The American Mathematical Monthly* 99, 5 (1992), 403–422.
Dexter Kozen. 1979. Semantics of Probabilistic Programs. In *FOCS*. 101–114.
Dexter Kozen. 1983. A Probabilistic PDL. In *STOC*. 291–297.
Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive proofs in higher-order concurrent separation logic. In *POPL*. 205–217.
Stephen Magill, Aleksandar Nanevski, Edmund Clarke, and Peter Lee. 2006. Inferring invariants in separation logic for imperative list-processing programs. *SPACE* 1, 1 (2006), 5–7.
Conrado Martínez and Salvador Roura. 1998. Randomized Binary Search Trees. *J. ACM* 45, 2 (1998), 288–323.
Annabelle McIver and Carroll Morgan. 2005. *Abstraction, Refinement and Proof for Probabilistic Systems.* Springer.
Annabelle McIver, Carroll Morgan, Benjamin Lucien Kaminski, and Joost-Pieter Katoen. 2018. A new proof rule for almost-sure termination. *PACMPL* 2, POPL (2018), 33:1–33:28. https://doi.org/10.1145/3158121
Carroll Morgan, Annabelle McIver, and Karen Seidel. 1996. Probabilistic Predicate Transformers. *Trans. on Programming Languages and Systems* 18, 3 (1996), 325–353.
Van Chan Ngo, Quentin Carbonneaux, and Jan Hoffmann. 2018. Bounded expectations: resource analysis for probabilistic programs. In *PLDI*. 496–512.
Peter W. O'Hearn. 2012. A Primer on Separation Logic (and Automatic Program Verification and Analysis). In *Software Safety and Security - Tools for Analysis and Verification.* 286–318.
Federico Olmedo, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. 2016. Reasoning about Recursive Probabilistic Programs. In *LICS*. 672–681.
William Pugh. 1990. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Commun. ACM* 33, 6 (1990), 668–676.
Martin Lee Puterman. 2005. *Markov Decision Processes: Discrete Stochastic Dynamic Programming.* John Wiley & Sons.
John Charles Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*. IEEE Computer Society, 55–74.

RIGHTSLINK

Dana Scott. 2008. The Algebraic Intepretation of Quantifiers. Intuitionistic and Classical. *Andrzej Mostowski and Foundational Studies* (2008), 289–312.

Joseph Tassarotti and Robert Harper. 2018. A Separation Logic for Concurrent Randomized Programs. *CoRR* abs/1802.02951 (2018). arXiv:1802.02951 http://arxiv.org/abs/1802.02951

Hongseok Yang and Peter W. O'Hearn. 2002. A Semantic Basis for Local Reasoning. In *FOSSACS*. 402–416.

RIGHTSLINK