Towards Concurrent Quantitative Separation Logic

Ira Fesefeldt ✉ 🏠
Software Modeling and Verification Group, RWTH Aachen University, Germany

Joost-Pieter Katoen ✉ 🏠
Software Modeling and Verification Group, RWTH Aachen University, Germany

Thomas Noll ✉ 🏠
Software Modeling and Verification Group, RWTH Aachen University, Germany

Abstract

In this paper, we develop a novel verification technique to reason about programs featuring concurrency, pointers and randomization. While the integration of concurrency and pointers is well studied, little is known about the combination of all three paradigms. To close this gap, we combine two kinds of separation logic – Quantitative Separation Logic and Concurrent Separation Logic – into a new separation logic that enables reasoning about lower bounds of the probability to realise a postcondition by executing such a program.

2012 ACM Subject Classification Theory of computation → Program verification; Theory of computation → Concurrent algorithms; Mathematics of computing → Probabilistic reasoning algorithms

Keywords and phrases Randomization, Pointers, Heap-Manipulating, Separation Logic, Concurrency

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2022.25

Related Version *Extended Version*: `https://arxiv.org/abs/2207.02822` [16]

1 Introduction and Related Work

In this paper, we aim to provide support for formal reasoning about concurrent imperative programs that are extended by two important features: dynamic data structures and randomisation. In other words, it deals with the analysis and verification of concurrent probabilistic pointer programs. This problem is of practical interest as many concurrent algorithms operating on data structures use randomisation to reduce the level of interaction between threads. For example, probabilistic skip lists [48] work well in the concurrent setting [17] because threads can independently manipulate nodes in the list without much synchronisation. In contrast, scalability of traditional balanced tree structures is difficult to achieve, since re-balancing operations may require locking access to large parts of the data structure. Bloom filters are another example of a probabilistic data structure supporting parallel access [8]. A further aspect is that stochastic modelling naturally arises when analysing faulty behaviour of (concurrent) software systems, as we later demonstrate in Section 5.

However, the combination of these features poses severe challenges when it comes to implementing and reasoning about concurrent randomised algorithms that operate on dynamic data structures. To give a systematic overview of related approaches, we mention that a number of program logics for reasoning about concurrent software have been developed [13, 14, 18, 30, 32, 43]. Next, we will address the programming-language extensions in isolation and then consider their integration. An overview is shown in Figure 1.

Pointers. Pointers constitute an essential concept in modern programming languages, and are used for implementing dynamic data structures like lists, trees etc. However, many software bugs can be traced back to the erroneous use of pointers by e.g. dereferencing null pointers or accidentally pointing to wrong parts of the heap, creating the need for

33rd International Conference on Concurrency Theory (CONCUR 2022).
Editors: Bartek Klin, Sławomir Lasota, and Anca Muscholl; Article No. 25; pp. 25:1–25:24
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

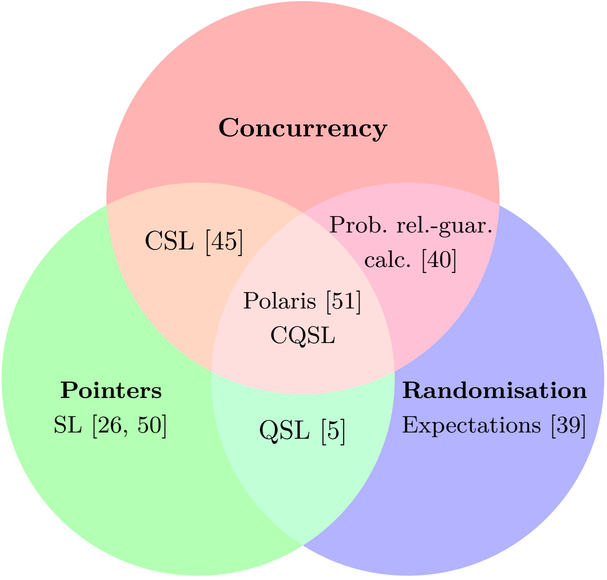

Figure 1 Overview of programming language features and formal approaches (CQSL denotes our concurrent extension of QSL).

computer-aided verification methods. The most popular formalism for reasoning about such programs is Separation Logic (SL) [26, 50], which supports Hoare-style verification of imperative, heap-manipulating and, possibly, concurrent programs. Its assertion language extends first-order logic with connectives that enable concise specifications of how program memory, or other resources, can be split-up and combined. In this way, SL supports local reasoning about the resources employed by programs. Consequently, program parts can be verified by considering only those resources they actually access – a crucial property for building scalable tools including automated verifiers [7, 28, 42, 47], static analysers [6, 10, 20], and interactive theorem provers [31].

The notion of resources, and in particular their controlled access, becomes even more important in a concurrent setting. Therefore, SL has been extended to Concurrent Separation Logic (CSL) [45] to enable reasoning about resource ownership, where the resource typically is dynamically allocated memory (i.e., the heap). The popularity of CSL is evident by the number of its extensions [9]. Of particular importance to our work is [53], which presents a soundness result for CSL that is formulated in an inductive manner, matching the "small-step" operational style of semantics. Here, we will employ a similar technique that also takes quantitative aspects (probabilities) into account.

Randomisation. Probabilistic programs (i.e., programs with the ability to sample from probability distributions) are increasingly popular for implementing efficient randomised algorithms [41] and describing uncertainty in systems [11, 19], among other similar tasks. In such applications, the purely qualitative (true vs. false) approach of classical logic is obviously not sufficient. The method advocated by us is based on weakest precondition reasoning as established in a classical setting by Dijkstra [12]. It has been extended to provide semantic foundations for probabilistic programs by Kozen [34, 35] and McIver & Morgan [39]. The latter also coined the term "weakest preexpectation" for random variables that take over the role of logical formulae when doing quantitative reasoning about probabilistic programs – the quantitative analogue of weakest preconditions. Their relation to operational models is studied in [21]. Moreover, weakest preexpectation reasoning has been shown to be useful for obtaining bounds on the expected resource consumption [44] and, especially, the expected run-time [33] of probabilistic programs.

However, verification techniques that support reasoning about both randomisation and dynamic data structures are rare – a surprising situation given that randomised algorithms typically rely on such data structures. One notable exception is the extension of SL to Quantitative Separation Logic (QSL) [4, 5], which marries SL and weakest preexpectations. QSL has successfully been applied to the verification of randomised algorithms, and QSL expectations have been formalised in Isabelle/HOL [24]. The present work builds on these results by additionally taking concurrency into account.

A prior program logic designed for reasoning about programs that are both concurrent and randomised but do not maintain dynamic data structures is the probabilistic rely-guarantee calculus developed by McIver et al. [40], which extends Jones's original rely-guarantee logic [30] by probabilistic constructs.

Later, Tassarotti & Harper [51] address the full setting of concurrent probabilistic pointer programs by combining CSL with probabilistic relational Hoare logic [3] to obtain Polaris, a Concurrent Separation Logic with support for probabilistic reasoning. Verification is thus understood as establishing a relation between a program to be analysed and a program which is known to be well-behaved. Programs which do not almost surely terminate, however, are outside the scope of their approach. In contrast, the goal of our method is to directly measure quantitative program properties on source-code level using weakest liberal preexpectations defined by a set of proof rules, including possibly non-almost surely terminating programs. Since the weakest liberal preexpectation includes non-termination probability, we can use invariants to bound the weakest liberal preexpectation of loops from below.

The main contributions of this paper are:

- the definition of a concurrent heap-manipulating probabilistic guarded command language (chpGCL) and its operational semantics in terms of Markov Decision Processes (MDP);
- a formal framework for reasoning about quantitative properties of chpGCL programs, which is obtained by extending classical weakest liberal preexpectations by resource invariants;
- a sound proof system that supports backward reasoning about such preexpectations; and
- the demonstration of our verification method on a (probabilistic) producer-consumer example.

The remainder of this paper is organised as follows. Section 2 introduces QSL as an assertion language for quantitative reasoning about (both sequential and concurrent) probabilistic pointer programs. In Section 3, we present the associated programming language (chpGCL) together with an operational semantics. Next, in Section 4 we develop a calculus for reasoning about lower bounds of weakest liberal preexpectations. Its usage is demonstrated in Section 5, and in Section 6 we conclude and explain further research directions. The paper is accompanied by an extended version providing elaborated proofs and an appendix providing additional details about the examples.

2 Quantitative Separation Logic

To reason about probability distributions over states of a program, we use Quantitative Separation Logic (QSL) [5, 37]. QSL is an extension of classical (or *qualitative*) Separation Logic in the sense that instead of mapping stack/heap pairs to booleans in order to gain a set characterization of states, we assign probabilities to stack/heap pairs.

▶ **Definition 2.1** (Stack). *Let* Vars *be a fixed set of variables. A stack* $s\colon \mathsf{Vars} \to \mathbb{Z}$ *is a mapping from variable symbols to values. We denote the set of all stacks by* Stacks*.*

When evaluating an (arithmetic or boolean) expression e with respect to a stack s, we write $e(s)$. In this sense, expressions are mappings from stacks to values. The stack that agrees with a stack s except for the value of x, which is mapped to v, is denoted as $s\,[x := v]$.

▶ **Definition 2.2** (Heaps). *A heap* $h\colon L \to \mathbb{Z}$ *is a mapping from a finite subset of locations* $L \subset \mathbb{N}_{>0}$ *to values. We denote the set of all heaps by* Heaps*.*
We furthermore write $\mathsf{dom}\,(h)$ *for the domain of* h*,* $h_1 \perp h_2$ *if and only if* $\mathsf{dom}\,(h_1) \cap \mathsf{dom}\,(h_2) = \emptyset$*, and for disjoint heaps* $h_1 \perp h_2$ *we define the disjoint union of heaps* h_1 *and* h_2 *as*

$$(h_1 \star h_2)(\ell) \quad = \quad \begin{cases} h_1(\ell) & \textit{if } \ell \in \mathsf{dom}\,(h_1) \\ h_2(\ell) & \textit{if } \ell \in \mathsf{dom}\,(h_2) \\ \textit{undef} & \textit{else}\ . \end{cases}$$

A pair of a *stack* and a *heap* is a *state* of the program. The stack is used to describe the variables of the program. The heap describes the addressable memory of the program.

▶ **Definition 2.3** (Program States). *A program state* $\sigma \in \mathsf{Stacks} \times \mathsf{Heaps}$ *is a pair consisting of a stack and a heap. The set of all states is denoted by* States*.*

Expectations are *random variables* that map states to non-negative reals. In this paper, we only consider one-bounded expectations. These do not map states to arbitrary non-negative reals, but only to reals between 0 and 1. The nomenclature of calling these expectations rather than random variables is due to the weakest preexpectation calculus being used to derive expectations.

▶ **Definition 2.4** (Expectations). *A (one-bounded) expectation* $X\colon \mathsf{States} \to [0, 1]$ *is a mapping from program states to probabilities. We write* $\mathbb{E}_{\leq 1}$ *for the set of all (one-bounded) expectations. We call an expectation* φ *qualitative if for all* $(s, h) \in \mathsf{States}$ *we have that* $\varphi(s, h) \in \{0, 1\}$*. We define the partial order* $(\mathbb{E}_{\leq 1}, \leq)$ *as the pointwise application of less than or equal, i.e.,* $X \leq Y$ *if and only if* $\forall (s, h) \in \mathsf{States}\ X(s, h) \leq Y(s, h)$*.*

We use capital letters for regular (one-bounded) expectations and Greek letters for qualitative expectations. As in [5], we choose to not give a specific syntax for QSL since the weakest liberal preexpectation of a given postexpectation – for which we provide more detail in Section 3 – may not be expressible in a given syntax. Instead, we prefer to interpret expectations as *extensional objects* that can be combined via various connectives. These connectives include (but are not limited to) the pointwise-applied connectives of addition, multiplication, exponentiation, maximum and minimum. As it is common in quantitative logics, the maximum/minimum is the quantitative extension of disjunction/conjunction, respectively. However, multiplication can be chosen as the quantitative extension of conjunction as well. We denote the substitution of a variable x by the expression e in the expectation X as $X\,[x := e]$ and define it as $X\,[x := e]\,(s, h) = X(s\,[x := e(s)]\,, h)$. When dealing with state predicates, we use Iverson brackets [27] to cast boolean values into integers:

$$[b]\,(s, h) = \begin{cases} 1 & \text{if } (s, h) \in b \\ 0 & \text{else} \end{cases}$$

Note that we could also define predicates as mappings from states to 0 or 1. We refrain from this, since (1) usage of Iverson brackets is standard in weakest preexpectation reasoning and (2) we may use QSL inside of Iverson brackets.

For a state (s, h), the empty heap predicate **emp** holds if and only if $\mathsf{dom}\,(h) = \emptyset$, the points-to predicate $e \mapsto e_0, \ldots, e_n$ holds if and only if $\mathsf{dom}\,(h) = \{e(s) + 0, \ldots, e(s) + n\}$ and $\forall i \in \{0, \ldots, n\}\ h(e(s) + i) = e_i(s)$, the allocated predicate $e \mapsto -$ holds if and only if $\mathsf{dom}\,(h) = \{e(s)\}$, and the equality predicate $e = e'$ holds if and only if $e(s) = e'(s)$.

We also use quantitative extensions of two separation connectives – the separating conjunction and the magic wand. The quantitative extension of the separating conjunction, which we call *separating multiplication*, maximises the value of the product of its arguments applied to separated heaps:

$$(X \star Y)(s, h) = \sup \{\, X(s, h_1) \cdot Y(s, h_2) \mid h_1 \star h_2 = h \,\}$$

The definition of separating multiplication is similar to the classical separating conjunction: the existential quantifier is replaced by a supremum and the conjunction by a multiplication. Note that the set over which the supremum ranges is never empty.

The (guarded) quantitative magic wand is defined for a qualitative first argument and a quantitative second argument. We minimise the value of the second argument applied to the original heap joined with a heap that evaluates the first argument to 1, i.e., for qualitative expectation φ and expectation Y we have:

$$(\varphi \mathrel{-\!\!\star} Y)(s, h) = \inf \{\, Y(s, h'') \mid \varphi(s, h') = 1,\ h'' = h \star h' \,\}$$

If the set is empty, the infimum evaluates to the greatest element of all probabilities, which is 1. Although it is also possible to allow expectations in both arguments (cf. [5]), we restrict ourselves to the guarded version of the magic wand. This restriction allows us to exploit the superdistributivity of multiplication, i.e., $\varphi \mathrel{-\!\!\star} (X \cdot Y) \geq (\varphi \mathrel{-\!\!\star} X) \cdot (\varphi \mathrel{-\!\!\star} Y)$.

▶ **Example 2.5.** To illustrate separating operations and lower bounding in QSL, we consider $X = [x \mapsto -] \star ([x \mapsto y] \mathrel{-\!\!\star} (0.5 \cdot [x \mapsto -]))$, $Y = 0.5 \cdot [x \mapsto -]$ and $Z = 0.5 \cdot [x \mapsto y]$. Let us consider the semantics of X in more detail. X is non-zero only for states that allocate exactly x. In this case, after changing the value pointed to by x to y, 0.5 is returned if x is still allocated (which obviously holds). Thus, the combination of separating multiplication and magic wand realises a change of value: First a pointer is removed from the heap by using separating multiplication, and afterwards we add it back with a different value using the magic wand. Note that $[x \mapsto y]$ is qualitative, which is required for our version of the magic wand. Then we have $X = Y$ and $Z \leq X$.

3 Programming Language and Operational Semantics

Our programming language is a concurrent extension of the heap-manipulating and probabilistic guarded command language [5]. Our language features both deterministic and probabilistic control flow, atomic regions, concurrent threads operating on shared memory, variable-based assignments, and heap manipulations. Although our language allows arbitrary shared memory, we will later only be able to reason about shared memory in the heap. Conditional choice without an else branch is considered syntactic sugar. Atomic regions consist of programs without memory allocation or concurrency. However, probabilistic choice is admitted. Programs that satisfy this restriction are called *tame*.

The reason to restrict the program fragment within atomic regions is that non-tame statements introduce non-determinism (as addresses to be allocated and schedulings of concurrent programs are chosen non-deterministically), which would increase the semantics' complexity while providing only little benefit (we refer to [1] regarding the handling of non-tame probabilistic programs in atomic regions). If an atomic region loops with a certain probability p, we instead transition to a non-terminating program with probability p.

▶ **Definition 3.1** (Concurrent Heap-Manipulating Probabilistic Guarded Command Language). *The concurrent heap-manipulating probabilistic guarded command language* chpGCL *is generated by the grammar*

$$
\begin{array}{rcll}
C & \longrightarrow & \downarrow & \textit{(terminated program)} \\
 & | & \texttt{diverge} & \textit{(non-terminating program)} \\
 & | & x \mathrel{\texttt{:=}} e & \textit{(assignment)} \\
 & | & \{\, C \,\}\ [\, e_p \,]\ \{\, C \,\} & \textit{(prob. choice)} \\
 & | & C \,;\, C & \textit{(seq. composition)} \\
 & | & \texttt{atomic}\,\{\, C \,\} & \textit{(atomic region)} \\
 & | & \texttt{if}\ (\, b \,)\ \{\, C \,\}\ \texttt{else}\ \{\, C \,\} & \textit{(conditional choice)} \\
 & | & \texttt{while}\,(\, b \,)\,\{\, C \,\} & \textit{(loop)} \\
 & | & C \parallel C & \textit{(concurrency)} \\
 & | & x \mathrel{\texttt{:=}} \texttt{new}\,(e_0, \ldots, e_n) & \textit{(allocation)} \\
 & | & \texttt{free}(e), & \textit{(disposal)} \\
 & | & x \mathrel{\texttt{:=}} \texttt{<}\, e \,\texttt{>} & \textit{(lookup)} \\
 & | & \texttt{<}\, e \,\texttt{>} \mathrel{\texttt{:=}} e' & \textit{(mutation)}
\end{array}
$$

where x is a variable, e, e', e_i: Stacks $\rightarrow \mathbb{Z}$ *are arithmetic expressions, e_p:* Stacks $\rightarrow [0,1]$ *is a probabilistic arithmetic expression and $b \subseteq$* Stacks *is a guard.*

▶ **Example 3.2.** We consider as running example a little program with two threads synchronizing over a randomised value:

$$
\begin{array}{l}
\texttt{<}\, r \,\texttt{>} \mathrel{\texttt{:=}} -1\,; \\
\quad \{\, \texttt{<}\, r \,\texttt{>} \mathrel{\texttt{:=}} 0 \,\}\ [0.5]\ \{\, \texttt{<}\, r \,\texttt{>} \mathrel{\texttt{:=}} 1 \,\}
\ \left\Vert\ \begin{array}{l} y \mathrel{\texttt{:=}} \texttt{<}\, r \,\texttt{>}\,; \\ \texttt{while}\,(\, y = -1 \,)\,\{\, y \mathrel{\texttt{:=}} \texttt{<}\, r \,\texttt{>} \,\}\ ; \end{array}\right.
\end{array}
$$

We first initialise our resource r with some integer that stands for an undefined value (here -1). The first thread now either assigns 0 or 1 with probability 0.5 to r. As soon as r has a new value, the second thread receives this value and terminates as r is not -1 any more.

We define the operational semantics of our programming language chpGCL in the form of a *Markov Decision Process* (MDP for short). An MDP allows the use of both *non-determinism*, which we need for interleaving multiple threads, and *probabilities*, which are used for encoding probabilistic program commands. A transition between states is thus always annotated with two parameters: (1) an action that is taken non-deterministically and (2) a probability to transition to a state given the aforementioned action.

▶ **Definition 3.3** (Markov Decision Process). *A Markov Decision Process $M = (U, Act, \mathbb{P})$ consists of a countable set of states U, a mapping from states to enabled actions $Act\colon U \rightarrow 2^A$ for a countable set of actions A, and a transition probability function $\mathbb{P}\colon (U \times A) \rightarrow U \rightarrow [0,1]$ where for all $\sigma \in U$ and $a \in Act(\sigma)$ we require $\sum_{\sigma' \in U} \mathbb{P}(\sigma, a)(\sigma') = 1$. We also use the shorthand notation $\sigma \xrightarrow[a]{p} \sigma'$ for $\mathbb{P}(\sigma, a)(\sigma') = p$ in case $p > 0$.*

$$\frac{}{x \;\texttt{:=}\; e, (s,h) \xrightarrow[\text{assign}]{1} \downarrow, (s\,[x \coloneqq e(s)]\,, h)}\ \text{ASSIGN}$$

$$\frac{e(s) \in \mathsf{dom}\,(h)}{x \;\texttt{:=}\; \texttt{<}\,e\,\texttt{>}, (s,h) \xrightarrow[\text{lookup}]{1} \downarrow, (s\,[x \coloneqq h(e(s))]\,, h)}\ \text{LOOKUP} \qquad \frac{e(s) \notin \mathsf{dom}\,(h)}{x \;\texttt{:=}\; \texttt{<}\,e\,\texttt{>}, (s,h) \xrightarrow[\text{lookup-abt}]{1} \texttt{abort}}\ \text{LOOKUP-ABT}$$

$$\frac{e(s) \in \mathsf{dom}\,(h)}{\texttt{<}\,e\,\texttt{>} \;\texttt{:=}\; e', (s,h) \xrightarrow[\text{mutation}]{1} \downarrow, (s, h\,[e(s) \coloneqq e'(s)])}\ \text{MUT} \qquad \frac{e(s) \notin \mathsf{dom}\,(h)}{\texttt{<}\,e\,\texttt{>} \;\texttt{:=}\; e', (s,h) \xrightarrow[\text{mutation-abt}]{1} \texttt{abort}}\ \text{MUT-ABT}$$

$$\frac{e(s) \in \mathsf{dom}\,(h) \qquad h' = h \setminus \{e(s) \mapsto h(e(s))\}}{\texttt{free}(e), (s,h) \xrightarrow[\text{free}]{1} \downarrow, (s, h')}\ \text{FREE} \qquad \frac{e(s) \notin \mathsf{dom}\,(h)}{\texttt{free}(e), (s,h) \xrightarrow[\text{free-abt}]{1} \texttt{abort}}\ \text{FREE-ABT}$$

$$\frac{\ell + 0, \ldots \ell + n \in \mathbb{N}_{>0} \setminus \mathsf{dom}\,(h) \qquad e_0(s) = v_0, \ldots, e_n(s) = v_n \qquad h' = h \star \{\ell + 0 \mapsto v_1\} \star \cdots \star \{\ell + n \mapsto v_n\}}{x \;\texttt{:=}\; \texttt{new}\,(e_0, \ldots, e_n)\,, (s,h) \xrightarrow[\text{alloc-}\ell]{1} \downarrow, (s\,[x \coloneqq \ell]\,, h')}\ \text{ALLOC}$$

Figure 2 Operational semantics of basic commands in chpGCL.

$$\frac{C_1, (s,h) \overset{p}{\underset{a}{\rightarrow}} C_1', (s',h')}{C_1 \,;\, C_2, (s,h) \overset{p}{\underset{a}{\rightarrow}} C_1' \,;\, C_2, (s',h')}\ \text{SEQ} \qquad \frac{C_2, (s,h) \overset{p}{\underset{a}{\rightarrow}} C_2', (s',h')}{\downarrow \,;\, C_2, (s,h) \overset{p}{\underset{a}{\rightarrow}} C_2', (s',h')}\ \text{SEQ-END}$$

$$\frac{C_1, (s,h) \overset{p}{\underset{a}{\rightarrow}} \texttt{abort}}{C_1 \,;\, C_2, (s,h) \overset{p}{\underset{a}{\rightarrow}} \texttt{abort}}\ \text{SEQ-ABT} \qquad \frac{C_2, (s,h) \overset{p}{\underset{a}{\rightarrow}} \texttt{abort}}{\downarrow \,;\, C_2, (s,h) \overset{p}{\underset{a}{\rightarrow}} \texttt{abort}}\ \text{SEQ-END-ABT}$$

$$\frac{s \in b}{\texttt{if}\ (\,b\,)\ \{\,C_1\,\}\ \texttt{else}\ \{\,C_2\,\}\,, (s,h) \xrightarrow[\text{if-t}]{1} C_1, (s,h)}\ \text{IF-T} \qquad \frac{s \notin b}{\texttt{if}\ (\,b\,)\ \{\,C_1\,\}\ \texttt{else}\ \{\,C_2\,\}\,, (s,h) \xrightarrow[\text{if-f}]{1} C_2, (s,h)}\ \text{IF-F}$$

$$\frac{s \in b}{\texttt{while}\,(\,b\,)\,\{\,C_1\,\}\,, (s,h) \xrightarrow[\text{loop-t}]{1} C_1 \,;\, \texttt{while}\,(\,b\,)\,\{\,C_1\,\}, (s,h)}\ \text{WHILE-T}$$

$$\frac{s \notin b}{\texttt{while}\,(\,b\,)\,\{\,C_1\,\}\,, (s,h) \xrightarrow[\text{loop-f}]{1} \downarrow, (s,h)}\ \text{WHILE-F} \qquad \frac{}{\texttt{diverge}, (s,h) \xrightarrow[\text{div}]{1} \texttt{diverge}, (s,h)}\ \text{DIV}$$

$$\frac{e_p(s) = p}{\{\,C_1\,\}\ [\,e_p\,]\ \{\,C_2\,\}\,, (s,h) \xrightarrow[\text{prob}]{p} C_1, (s,h)}\ \text{PROB-L} \qquad \frac{e_p(s) = p}{\{\,C_1\,\}\ [\,e_p\,]\ \{\,C_2\,\}\,, (s,h) \xrightarrow[\text{prob}]{1-p} C_2, (s,h)}\ \text{PROB-R}$$

Figure 3 Operational semantics of non-concurrent control-flow operations in chpGCL.

We define the operational semantics of chpGCL as an MDP. A state in this MDP consists of a chpGCL program to be executed and a program state (s, h). The meaning of basic commands, i.e., assignments, heap mutations, heap lookups, memory allocation and disposal, is defined by the inference rules shown in Figure 2. An action is enabled if and only if an inference rule for this action exists. We use an `abort` keyword to indicate that a memory access error happened and terminate at this state. We consider aborted runs as undesired runs. The condition $\sum_{\sigma' \in U} \mathbb{P}(\sigma, a)(\sigma') = 1$ holds for all states in a chpGCL program. States with program $\downarrow$ or `abort` have no enabled actions, thus the condition holds trivially for all enabled actions a; non-probabilistic programs only have actions with trivial distributions; states with probabilistic choice only have a single action with a biased coin-flip distribution; and other programs are composed of these.

Control-flow statements include while loops, conditional choice, sequential composition and probabilistic choice, and we define their operational semantics in Figure 3. For the sake of brevity, we do not include a command to sample from a distribution.

The remaining control-flow statements handle concurrency, i.e., the concurrent execution of two threads and the atomic execution of regions. An atomic region may only terminate with a certain probability. The notation $C, (s,h) \overset{p}{\rightarrow}{}^{*}\ \ldots$ denotes that program C does not terminate on state (s,h) with probability p. As mentioned before, we will only allow

$$\frac{C,(s,h) \overset{p}{\rightarrow}{}^{*} \downarrow,(s',h') \qquad C \text{ is tame}}{\texttt{atomic}\,\{\,C\,\},(s,h) \xrightarrow[\text{atomic}]{p} \downarrow,(s',h')}\ \text{ATOM-END} \qquad \frac{C,(s,h) \overset{p}{\rightarrow}{}^{*} \texttt{abort} \qquad C \text{ is tame}}{\texttt{atomic}\,\{\,C\,\},(s,h) \xrightarrow[\text{atomic}]{p} \texttt{abort}}\ \text{ATOM-ABT}$$

$$\frac{C,(s,h) \overset{p}{\rightarrow}{}^{*} \ldots \qquad C \text{ is tame}}{\texttt{atomic}\,\{\,C\,\},(s,h) \xrightarrow[\text{atomic}]{p} \texttt{diverge},(s,h)}\ \text{ATOM-LOOP}$$

$$\frac{C_1,(s,h) \overset{p}{\underset{a}{\rightarrow}} C_1',(s',h')}{C_1 \parallel C_2\,,(s,h) \xrightarrow[C1,a]{p} C_1' \parallel C_2\,,(s',h')}\ \text{CON-L} \qquad \frac{C_2,(s,h) \overset{p}{\underset{a}{\rightarrow}} C_2',(s',h')}{C_1 \parallel C_2\,,(s,h) \xrightarrow[C2,a]{p} C_1 \parallel C_2'\,,(s',h')}\ \text{CON-R}$$

$$\frac{C_1,(s,h) \overset{p}{\underset{a}{\rightarrow}} \texttt{abort}}{C_1 \parallel C_2\,,(s,h) \xrightarrow[C1,a]{p} \texttt{abort}}\ \text{CON-L-ABT} \qquad \frac{C_2,(s,h) \overset{p}{\underset{a}{\rightarrow}} \texttt{abort}}{C_1 \parallel C_2\,,(s,h) \xrightarrow[C2,a]{p} \texttt{abort}}\ \text{CON-R-ABT}$$

$$\frac{}{\downarrow \parallel \downarrow\,,(s,h) \xrightarrow[\text{con-end}]{1} \downarrow,(s,h)}\ \text{CON-END}$$

Figure 4 Operational semantics of concurrent control-flow operations in chpGCL.

tame programs inside atomic regions. A tame program does not require any (scheduling) actions since its Markov model is fully probabilistic. To formally define the syntax used in the inference rules for atomic regions, we first need to introduce *schedulers*, which are used to resolve non-determinism in an MDP. There are various classes of schedulers, and indeed we will later allow the use of different classes. However, we do require that all schedulers are deterministic and may have a history. This especially rules out any randomised scheduler, which would be an interesting topic, but is out of scope for the results presented here. Our schedulers use finite sequences of MDP states as histories.

▶ **Definition 3.4** (Scheduler). *A scheduler is a mapping $\mathfrak{s}\colon U^+ \to A$ from histories of states to enabled actions, i.e., $\mathfrak{s}(\sigma_1 \ldots \sigma_n) \in Act(\sigma_n)$. We denote the set of all schedulers by $\mathbb{S}$.*

For final states σ' (i.e., with program $\downarrow$ or `abort`) and an MDP $(U, \mathrm{Act}, \mathbb{P})$, we define

$$\mathrm{reach}(n, \sigma_1, \mathfrak{s}, \sigma') = \sum \left[\prod_{i=1}^{m-1} \mathbb{P}(\sigma_i, \mathfrak{s}(\sigma_1 \ldots \sigma_i))(\sigma_{i+1}) \;\middle|\; \sigma_1 \ldots \sigma_m \in U^m, \sigma_m = \sigma', m \le n \right], \tag{1}$$

$$\sigma \overset{p}{\underset{\mathfrak{s}}{\rightarrow}}{}^{*} \sigma' \quad \text{iff} \quad p = \lim_{n\to\infty} \mathrm{reach}(n, \sigma, \mathfrak{s}, \sigma')\ , \tag{2}$$

$$\sigma \xrightarrow[\mathfrak{s}]{1-p}{}^{*} \ldots \quad \text{iff} \quad p = \sum_{\sigma' \text{ final}} \lim_{n\to\infty} \mathrm{reach}(n, \sigma, \mathfrak{s}, \sigma')\ . \tag{3}$$

For a function f and a predicate b, we write $[f(x) \mid x \in b]$ for the bag consisting of the values $f(x)$ with $x \in b$. We use notation (1) to calculate the probability to reach the final state σ' from σ_1 in at most n steps w.r.t $\mathfrak{s}$. We unroll the MDP here into the Markov Chain induced by $\mathfrak{s}$ after at most n steps (cf. [2, Definition 10.92]). With notation (2), we define the reachability probability of a final state and with notation (3), we define the probability of non-termination. We avoid reasoning about uncountable sets of paths in case of non-termination by taking the probability to not reach a final state, i.e., a state with program $\downarrow$ or `abort`. A scheduler $\mathfrak{s}$ is unique if for every state $\sigma \in U$ there is at most one enabled action $\mathfrak{s}$ can map to, i.e., $|\mathrm{Act}(\sigma)| \le 1$. In that case, we usually omit the corresponding transition label.

To reason about the operational semantics using QSL, we use weakest *liberal* preexpectations [5, 38], which take the greatest lower bound of the expected value with respect to a postexpectation together with the probability of non-termination for all schedulers that we want to consider. We allow subsets of schedulers $S \subseteq \mathbb{S}$ in order to apply fairness conditions. Later, we only consider the complete set of schedulers. In that case, we omit the superscript from the function wlp, which is defined in the following.

▶ **Definition 3.5** (Weakest Liberal Preexpectation)**.** *For a program C and an expectation X, we define the weakest liberal preexpectation with respect to a set of schedulers $\emptyset \neq S \subseteq \mathbb{S}$ as*

$$\begin{aligned}\mathsf{wlp}^{S}[\![C]\!]\,(X)\,(s,h) \quad &= \quad \inf \Big\{ \sum \Big[p \cdot X(s',h') \mid C,(s,h) \xrightarrow[\mathfrak{s}]{p}{}^{*} \ \downarrow,(s',h') \Big] + p_{div} \\ &\qquad\qquad \Big|\ \mathfrak{s} \in S \text{ and } C,(s,h) \xrightarrow[\mathfrak{s}]{p_{div}}{}^{*} \ \ldots \Big\}\ .\end{aligned}$$

▶ **Example 3.6.** For program C in Example 3.2, we evaluate (without proof) $\mathsf{wlp}[\![C]\!]\,([y=0]) = \mathsf{wlp}^{\mathbb{S}}[\![C]\!]\,([y=0]) = 0.5 \star [r \mapsto -]$. That is, if r is allocated, then the likelihood of C terminating without aborting in a state in which y equals 0 is 0.5, and zero otherwise. We will prove that this is a lower bound in Example 4.2.

4 Weakest Safe Liberal Preexpectations

For sequential probabilistic programs, a backwards expectation transformer can be defined to compute wlp [5]. This is not feasible for concurrent programs due to the non-locality of shared memory. Instead, we drop exact computation in our approach and reason about lower bounds of wlp by using inference rules similar to Hoare triples. To support shared memory, we furthermore introduce a modified version of wlp – the weakest *resource-safe* liberal preexpectation. The general idea as inspired by [53] is to prove that the shared memory is invariant with respect to a qualitative expectation, which we call a *resource invariant.* In other words, the shared memory is proven to be *safe* with respect to the resource invariant. We archive this by enforcing that at every point in the program's execution (except for executions in atom regions), some part of the heap is satisfied by the resource invariant. In Example 4.2 we use the resource invariant $\max\{\,[r \mapsto 0],\ [r \mapsto -1]\,\}$ to prove the lower bound from Example 3.6. We enforce that the program states do not include the shared memory any more, the transitions however are taken with any possible shared memory.

▶ **Definition 4.1** (Weakest Resource-Safe Liberal Preexpectation)**.** *We first consider the expectation after one step with respect to a mapping from programs to expectations, that is, for a program C and a mapping $t\colon \mathsf{chpGCL} \to \mathbb{E}_{\leq 1}$, we define*

$$\begin{aligned}\mathsf{step}[\![C]\!]\,(t)\,(s,h) \quad &= \quad \inf \Big\{ \sum \Big[p \cdot t(C')(s',h') \mid C,(s,h) \xrightarrow[a]{p} C',(s',h') \Big] \\ &\qquad\qquad \Big|\ a \in \mathit{Act}(C,(s,h)) \Big\}\ .\end{aligned}$$

We define the weakest resource-safe liberal preexpectation after n steps for a program C, a postexpectation X and a (qualitative) resource invariant ξ as

$$\mathsf{wrlp}_{n}[\![C]\!]\,(X \mid \xi) = \begin{cases} 1 & \text{if } n = 0 \\ X & \text{if } n \neq 0 \text{ and } C = \downarrow \\ \xi \mathrel{-\!\!\star} \mathsf{step}[\![C]\!]\,\big(\lambda C'.\ \mathsf{wrlp}_{n-1}[\![C']\!]\,(X \mid \xi) \star \xi\big) & \text{otherwise}^{1}. \end{cases}$$

[1] We use $\lambda C'.\ X$ for the function which, when applied to the argument C, reduces to X in which every occurrence of C' in X is replaced by C.

Finally, we define the weakest resource-safe liberal preexpectation for arbitrarily many steps as

$$\mathsf{wrlp}[\![C]\!]\,(X \mid \xi) = \lim_{n\to\infty} \mathsf{wrlp}_n[\![C]\!]\,(X \mid \xi) \enspace .$$

An important observation is that for the special resource invariant [**emp**], wlp and wrlp coincide (cf. [16]). This enables us to reason about lower bounds for probabilities of qualitative preconditions (and in general lower bounds for the expected value of one-bounded random variables). When reasoning about such probabilities, we first express a property for which we aim to prove a lower bound on wlp, afterwards we can transform it into wrlp with the resource invariant [**emp**] and use special rules to enrich the resource invariant with more information. The resource invariant should always cover all possible states that the shared memory may be in at any time during the program's execution. It is fine if the resource invariant is violated during executions of atomic regions, since we only care about safeness during executions with inferences between threads.

We mention that wrlp is heavily inspired by [53]. We formalise the connection between Vafeiadis' Concurrent Separation Logic and our weakest resource-safe liberal preexpectation below. In [53] a judgement is defined by a safe predicate that is similar to how we defined wrlp.

▶ **Definition 4.1** (Safe Judgements [53]). *The predicate $\mathit{safe}_n(C, s, h, \xi, \varphi)$ holds for qualitative φ and ξ and non-probabilistic program C if and only if*

1. *if $n = 0$, then it holds always; and*
2. *if $n > 0$ and $C = \downarrow$, then $\varphi(s, h) = 1$; and*
3. *if $n > 0$ and for all h_ξ and h_F with $\xi(s, h_\xi) = 1$ and $h \perp h_\xi \perp h_F$, then for all enabled actions $a \in \mathit{Act}(C, (s, h \star h_\xi \star h_F))$ we do not have $C, (s, h \star h_\xi \star h_F) \xrightarrow[a]{1}$ `abort`; and*
4. *if $n > 0$ and for all h_ξ, h_F, C', s and h, with $\xi(s, h_\xi) = 1$, and $h \perp h_\xi \perp h_F$, and $C, (s, h \star h_\xi \star h_F) \xrightarrow[a]{1} C', (s', h')$, then there exists h'' and h'_ξ such that $h' = h'' \star h'_\xi \star h_F$ and $\xi(s', h'_\xi) = 1$ and $\mathit{safe}_{n-1}(C', s', h'', \xi, \varphi)$.*

For qualitative φ, ψ and ξ, we say that $\xi \models \{\psi\}\ C\ \{\varphi\}$ holds if and only if for all stack/heap pairs s, h the statement $\psi(s, h) = 1 \Rightarrow \forall n \in \mathbb{N}.\ \mathit{safe}_n(C, s, h, \xi, \varphi)$ holds.

A program C is framing enabled[2] if we can always extend the heap without changing the behaviour of C.

▶ **Definition 4.2** (Framing Enabledness). *A non-probabilistic program C is framing enabled if for all heaps h_F with $h \perp h_F$ and all enabled actions $a \in \mathit{Act}(C, (s, h \star h_F))$, it holds: if $C, (s, h) \xrightarrow[a]{1} C', (s', h')$, then also $C, (s, h \star h_F) \xrightarrow[a]{1} C', (s', h' \star h_F)$.*

The next theorem states that wrlp is a conservative extension of safe.

▶ **Theorem 4.1** (Conservative Extension of Concurrent Separation Logic). *For a framing-enabled non-probabilistic program C and qualitative expectations φ, ψ and ξ, we have*

$$\varphi \;\leq\; \mathsf{wrlp}[\![C]\!]\,(\psi \mid \xi) \quad \textit{iff} \quad \xi \models \{\varphi\}\ C\ \{\psi\} \enspace .$$

Proof. See [16]. ◀

[2] Indeed every non-probabilistic chpGCL program is framing enabled (cf. [16]).

$$\frac{}{X \;\leq\; \mathsf{wrlp}\llbracket \downarrow \rrbracket\,(X \mid \xi)}\ \text{term} \qquad\qquad \frac{Y \leq X\,[x := e]}{Y \;\leq\; \mathsf{wrlp}\llbracket x\ \mathtt{:=}\ e\rrbracket\,(X \mid \xi)}\ \text{assign}$$

$$\frac{Y \leq \sup_{v\in\mathbb{Z}}\,[e \mapsto v] \star ([e \mapsto v] \mathrel{-\!\!\star} X\,[x := v])}{Y \;\leq\; \mathsf{wrlp}\llbracket x\ \mathtt{:=}\ \mathtt{<}\,e\,\mathtt{>}\rrbracket\,(X \mid \xi)}\ \text{look} \qquad \frac{Y \leq [e \mapsto\ -\,] \star ([e \mapsto e'] \mathrel{-\!\!\star} X)}{Y \;\leq\; \mathsf{wrlp}\llbracket \mathtt{<}\,e\,\mathtt{>}\ \mathtt{:=}\ e'\rrbracket\,(X \mid \xi)}\ \text{mut}$$

$$\frac{Y \leq \inf_{v\in\mathbb{Z}}\,[v \mapsto e_1,\ldots,e_n] \mathrel{-\!\!\star} X\,[x := v]}{Y \;\leq\; \mathsf{wrlp}\llbracket x\ \mathtt{:=}\ \mathtt{new}\,(e_1,\ldots,e_n)\rrbracket\,(X \mid \xi)}\ \text{alloc} \qquad \frac{Y \leq X \star [x \mapsto\ -\,]}{Y \;\leq\; \mathsf{wrlp}\llbracket \mathtt{free}(x)\rrbracket\,(X \mid \xi)}\ \text{disp}$$

Figure 5 Proof rules for wrlp for basic commands.

$$\frac{X \;\leq\; \mathsf{wrlp}\llbracket C_1\rrbracket\,(Y \mid \xi) \qquad Y \;\leq\; \mathsf{wrlp}\llbracket C_2\rrbracket\,(Z \mid \xi)}{X \;\leq\; \mathsf{wrlp}\llbracket C_1\,;\,C_2\rrbracket\,(Z \mid \xi)}\ \text{seq}$$

$$\frac{X_1 \;\leq\; \mathsf{wrlp}\llbracket C_1\rrbracket\,(Y \mid \xi) \qquad X_2 \;\leq\; \mathsf{wrlp}\llbracket C_2\rrbracket\,(Y \mid \xi)}{[b]\cdot X_1 + [\neg b]\cdot X_2 \;\leq\; \mathsf{wrlp}\llbracket \mathtt{if}\ (\,b\,)\ \{\,C_1\,\}\ \mathtt{else}\ \{\,C_2\,\}\rrbracket\,(Y \mid \xi)}\ \text{if}$$

$$\frac{I \leq [b]\cdot X + [\neg b]\cdot Y \qquad X \;\leq\; \mathsf{wrlp}\llbracket C\rrbracket\,(I \mid \xi)}{I \;\leq\; \mathsf{wrlp}\llbracket \mathtt{while}\,(\,b\,)\,\{\,C\,\}\rrbracket\,(Y \mid \xi)}\ \text{while} \qquad \frac{}{X \;\leq\; \mathsf{wrlp}\llbracket \mathtt{diverge}\rrbracket\,(Y \mid \xi)}\ \text{div}$$

$$\frac{X_1 \;\leq\; \mathsf{wrlp}\llbracket C_1\rrbracket\,(Y \mid \xi) \qquad X_2 \;\leq\; \mathsf{wrlp}\llbracket C_2\rrbracket\,(Y \mid \xi)}{e_p\cdot X_1 + (1-e_p)\cdot X_2 \;\leq\; \mathsf{wrlp}\llbracket\{\,C_1\,\}\,[\,e_p\,]\,\{\,C_2\,\}\rrbracket\,(Y \mid \xi)}\ \text{p-choice} \qquad \frac{X \;\leq\; \mathsf{wrlp}\llbracket C\rrbracket\,(Y \star \xi \mid [\mathbf{emp}])}{X \;\leq\; \mathsf{wrlp}\llbracket \mathtt{atomic}\,\{\,C\,\}\rrbracket\,(Y \mid \xi)}\ \text{atomic}$$

$$\frac{X \;\leq\; \mathsf{wrlp}\llbracket C\rrbracket\,(Y \mid \xi \star \pi)}{X \star \pi \;\leq\; \mathsf{wrlp}\llbracket C\rrbracket\,(Y \star \pi \mid \xi)}\ \text{share}$$

$$\frac{X_1 \;\leq\; \mathsf{wrlp}\llbracket C_1\rrbracket\,(Y_1 \mid \xi) \qquad X_2 \;\leq\; \mathsf{wrlp}\llbracket C_2\rrbracket\,(Y_2 \mid \xi) \qquad \forall i \in \{1,2\}\ \mathrm{Write}(C_i) \cap \mathrm{Vars}\,(C_{3-i}, Y_{3-i}, \xi) = \emptyset}{X_1 \star X_2 \;\leq\; \mathsf{wrlp}\llbracket\, C_1 \parallel C_2\,\rrbracket\,(Y_1 \star Y_2 \mid \xi)}\ \text{concur}$$

Figure 6 Proof rules for wrlp for control-flow commands.

We define wrlp inductively by means of a number of inference rules. We do not use classic Hoare triples due to difficulties arising when interpreting a wrlp statement forward. These difficulties are due to Jones's counterexample [29, p. 135]: Given the constant preexpectation 0.5 and the program C: $\{\,x\ \mathtt{:=}\ 0\,\}\ [\,0.5\,]\ \{\,x\ \mathtt{:=}\ 1\,\}$, what is the postexpectation? Two possible answers are $0.5 = \mathsf{wlp}\llbracket C\rrbracket\,([x = 0])$ and $0.5 = \mathsf{wlp}\llbracket C\rrbracket\,([x = 1])$, but a combination of both is not possible. For this reason, we highlight the backwards interpretation of our judgements by writing them as $X \;\leq\; \mathsf{wrlp}\llbracket C\rrbracket\,(Y \mid \xi)$, where X is a (lower bound for the weakest liberal) preexpectation, C is the program, Y is the postexpectation and ξ is the resource invariant.

For basic commands, as shown in Figure 5, we can just re-use the QSL proof rules for weakest liberal preexpectations (wlp) of non-concurrent programs, as given in [5]. However, for wrlp these proof rules only allow lower bounding the preexpectation since we do not want to reason about the resource invariant if not necessary.

For commands handling control flow, as shown in Figure 6, we use mostly standard rules. Atomic regions regain access to the resource invariant. The share rule allows us to enrich the resource invariant. The rule for concurrency enforces that only local variables or read-only variables are used in each thread. One could as well allow shared variables that are owned by the resource invariant. However, for the sake of brevity we do not include this here.

We also introduce several proof rules that make reasoning easier, see Figure 7. A program is *almost surely terminating* with respect to a set of schedulers if the program terminates with probability one for every initial state and every scheduler in this set. Even though there is a plethora of work on almost-sure termination for (sequential) probabilistic programs (cf. [25] for an overview), techniques for checking almost-sure termination in a concurrent setting are sparse [22, 23, 36, 52]. Here, interpreting probabilistic choice as non-determinism and proving sure termination instead using techniques such as [15, 49] is an alternative.

$$\frac{X' \leq \mathsf{wlp}^S\llbracket C\rrbracket\,(X) \qquad Y' \leq \mathsf{wlp}^S\llbracket C\rrbracket\,(Y) \qquad C \text{ is AST w.r.t. } S \qquad a \in \mathbb{R}_{\geq 0}}{a \cdot X' + Y' \leq \mathsf{wlp}^S\llbracket C\rrbracket\,(a \cdot X + Y)}\ \text{superlin}$$

$$\frac{X \ \leq\ \mathsf{wrlp}\llbracket C\rrbracket\,(Y \mid [\mathbf{emp}])}{X \leq \mathsf{wlp}^S\llbracket C\rrbracket\,(Y)}\ \mathsf{wlp\text{-}wrlp} \qquad \frac{X \ \leq\ \mathsf{wrlp}\llbracket C\rrbracket\,(Y \mid \xi) \qquad \mathrm{Write}(C) \cap \mathrm{Vars}\,(Z) = \emptyset}{X \star Z \ \leq\ \mathsf{wrlp}\llbracket C\rrbracket\,(Y \star Z \mid \xi)}\ \text{frame}$$

$$\frac{X \star \xi \ \leq\ \mathsf{wrlp}\llbracket C\rrbracket\,(Y \star \xi \mid [\mathbf{emp}]) \qquad C \text{ is a terminating atom}}{X \ \leq\ \mathsf{wrlp}\llbracket C\rrbracket\,(Y \mid \xi)}\ \text{atom}$$

$$\frac{X \leq X' \qquad X' \ \leq\ \mathsf{wrlp}\llbracket C\rrbracket\,\big(Y' \mid \xi\big) \qquad Y' \leq Y}{X \ \leq\ \mathsf{wrlp}\llbracket C\rrbracket\,(Y \mid \xi)}\ \text{monotonic}$$

$$\frac{X \ \leq\ \mathsf{wrlp}\llbracket C\rrbracket\,(Y \mid \xi) \qquad X' \ \leq\ \mathsf{wrlp}\llbracket C\rrbracket\,\big(Y' \mid \xi\big)}{\max\left\{\, X,\ X' \,\right\} \ \leq\ \mathsf{wrlp}\llbracket C\rrbracket\,\big(\max\left\{\, Y,\ Y' \,\right\} \mid \xi\big)}\ \text{max}$$

$$\frac{X \ \leq\ \mathsf{wrlp}\llbracket C\rrbracket\,(Y \mid \xi) \qquad X' \ \leq\ \mathsf{wrlp}\llbracket C\rrbracket\,\big(Y' \mid \xi\big) \qquad \xi \text{ precise}}{\min\left\{\, X,\ X' \,\right\} \ \leq\ \mathsf{wrlp}\llbracket C\rrbracket\,\big(\min\left\{\, Y,\ Y' \,\right\} \mid \xi\big)}\ \text{min}$$

$$\frac{X \ \leq\ \mathsf{wrlp}\llbracket C\rrbracket\,(Y \mid \xi) \qquad X' \ \leq\ \mathsf{wrlp}\llbracket C\rrbracket\,\big(Y' \mid \xi\big) \qquad \xi \text{ precise} \qquad \mathrm{Write}(C) \cap \mathrm{Vars}\,(e) = \emptyset}{e \cdot X + (1-e) \cdot X' \ \leq\ \mathsf{wrlp}\llbracket C\rrbracket\,\big(e \cdot Y + (1-E) \cdot Y' \mid \xi\big)}\ \text{convex}$$

Figure 7 Auxiliary proof rules for wrlp.

The first rule in Figure 7 uses superlinearity to split a given postexpectation into a sum of postexpectations, for which proving a lower bound on the preexpectation might be easier. We only allow the use of superlinearity for wlp (and not for wrlp) because we need a restricted set of schedulers to enforce fairness conditions. Fairness conditions are required to reason about termination for concurrent programs with some sort of blocking behaviour. We are then able to transform wlp into wrlp by using the wlp-wrlp rule. Whether wrlp can also be defined with fairness conditions in mind and thus applying superlinearity directly on wrlp, is an open question. The frame rule is of central importance to the Separation Logic approach, as it supports local reasoning about only the relevant part of the heap [46]. The atom rule can be used similarly to the rule for atomic regions. Monotonicity is the quantitative version of the rule of consequence and is used to reduce and increase the post- and preexpectation respectively. The max, min and convex rules eliminate max, min and convex sum operations, respectively. The min and convex rule require preciseness of the resource invariant – similarly to how [53] required preciseness for the conjunction rule. An expectation is *precise* if for any stack there is at most one heap for which the expectation is not zero. For the min rule, this is not surprising as the minimum behaves like conjunction in case of qualitative expectations. Requiring preciseness also for the convex rule is due to the missing superlinearity of the separating multiplication for non-precise expectations.

▶ **Theorem 4.2** (Soundness of proof rules). *For every proof rule in Figures 5–7 it holds that if their premises hold, the conclusion holds as well.*

Proof. See [16]. ◀

▶ **Example 4.2.** We are now able to establish the lower bound computed in Example 3.6 using the proof rules. Instead of constructing a proof tree by composing inference rules, we annotate program locations with their respective pre- and postexpectations. The interpretation is standard; for preexpectation X, postexpectation Y, resource invariant ξ and program C:

$$\begin{array}{lcl} /\!\!/\ \ X \ \mid\ \xi & & \\ C & \quad\text{iff}\quad & X \ \leq\ \mathsf{wrlp}\llbracket C\rrbracket\,(Y \mid \xi) \\ /\!\!/\ \ Y \ \mid\ \xi & & \end{array}$$

Proofs in this style should only be read backwards from bottom to top. They will not include applications of the proof rules for atomic programs and of the share rule as this may lead to incorrect interpretations. For our example, we use the resource invariant $\xi = \max\{[r \mapsto 0], [r \mapsto -1]\}$, which we guessed by collecting all possible values stored in location r during executions yielding our postexpectation. We assume that the memory of the initial heap h only contains a single location r with value -1. This assumption is reflected by the resource invariant and will only allow us to reason about executions with such an initial heap. The other possible value for the location r is 0, since the left program may mutate the heap. Indeed, there are also executions where the value of location r is 1. For these, the program only terminates in states violating the postexpectation $[y = 0]$. We can further show:

$$
\begin{array}{l}
/\!\!/\ \ 0.5 \star 1 \ \mid\ \xi \\
\begin{array}{l||l}
 & /\!\!/\ \ 1 \ \mid\ \xi \\
/\!\!/\ \ 0.5 \ \mid\ \xi & y \mathrel{\texttt{:=}} \texttt{<}\,r\,\texttt{>}\,; \\
\{\,\texttt{<}\,r\,\texttt{>} \mathrel{\texttt{:=}} 0\,\}\,[\,0.5\,]\,\{\,\texttt{<}\,r\,\texttt{>} \mathrel{\texttt{:=}} 1\,\} & /\!\!/\ \ \max\{[y=0],\ [y=-1]\} \ \mid\ \xi \\
/\!\!/\ \ 1 \ \mid\ \xi & \texttt{while}\,(\,y=-1\,)\,\{\,y \mathrel{\texttt{:=}} \texttt{<}\,r\,\texttt{>}\,\}\ ; \\
 & /\!\!/\ \ [y=0] \ \mid\ \xi
\end{array} \\
/\!\!/\ \ 1 \star [y=0] \ \mid\ \xi
\end{array}
$$

To handle concurrency, we separate our postexpectation into the expectation 1 for the left program and the expectation $[y = 0]$ for the right program. The left program includes a probabilistic choice, for which we use the atom rule to infer that the preexpectation is 1 in the left branch of the probabilistic choice, as the resource invariant allows mutating the value of location r to 1 and the resource invariant can be re-established since we can lower bound all possible values for the location r that are not -1 or 0 to zero. Moreover, we lower bound the right branch of the probabilistic choice by zero, because zero is a lower bound of any expectation. The right program iterates until the value of location r has been mutated. Our resource invariant contains all possible values that the program can expect here. For the loop invariant, we connect all possible values of y using a disjunction over 0 and -1, as we disregard executions where $y = 1$. Lastly, we can apply the loop invariant to the lookup of r and since this matches our resource invariant, the resulting preexpectation is one.

Thus, we have established that $0.5 \ \leq\ \mathsf{wrlp}[\![C]\!]\,([y = 0] \mid \xi)$. Using the share rule we can further infer that $0.5 \star \xi \ \leq\ \mathsf{wrlp}[\![C]\!]\,([y = 0] \star \xi \mid [\mathbf{emp}])$. Lastly, we can clean up the statement using monotonicity and the wlp-wrlp rule to obtain $0.5 \star \xi \leq \mathsf{wlp}[\![C]\!]\,([y = 0])$. For details on the probabilistic choice and the loop invariant, we refer to Appendix A.1.

5 Example: A Producer, a Consumer and a Lossy Channel

A producer-consumer system is often used when presenting verification techniques for concurrent programs. We continue this tradition, extending this example by probabilistic elements, see Figure 8. Video and audio streaming is an example for such a system, where data losses are acceptable if they do not exceed a certain limit. Moreover, by enriching the resource invariant with a predicate defining an appropriate data structure, this example can be used as a template to reason about systems communicating using a shared data structure. We consider a producer that randomly generates data (1 or 2) and stores it in an array of

```
$l$ := 0;
$y_1, y_2, y_3$ := $k$;

while ($y_1 \geq 0$) {
  { $x_1$ := 1 } [0.5] { $x_1$ := 2 } ;
  <$z_1 + y_1$> := $x_1$ ;
  $y_1$ := $y_1 - 1$
}

||

while ($y_2 \geq 0$) {
  $x_2$ := <$z_1 + y_2$> ;
  if ($x_2 \neq 0$) {
      {<$z_2 + y_2$> := $x_2$}
    [$p$]
      {<$z_2 + y_2$> := $-1$} ;
    $y_2$ := $y_2 - 1$
  }
}

||

while ($y_3 \geq 0$) {
  $x_3$ := <$z_2 + y_3$> ;
  if ($x_3 \neq 0$) {
    if ($x_3 \neq -1$) {$l$ := $l + 1$} ;
    $y_3$ := $y_3 - 1$
  }
}
```

Figure 8 A program consisting of the tree threads: a producer (left), a consumer (right) and lossy channel (middle) for communication between the prior threads.

size k indexed by z_1. The data has to be transferred to a consumer. However, the consumer does not have direct access to the array maintained by the producer. Instead a third party, the lossy channel, transfers data from the array maintained by the producer to a different k-sized array that is indexed by z_2, and that can be accessed by the consumer. However, the channel is not reliable. With a probability of $1 - p$, it loses a value and instead stores invalid data (encoded as -1) at the respective array position. The consumer discards invalid data and counts in l how many valid elements it received until all array elements have been attempted to be transmitted once. For the sake of brevity, we leave out the allocation of the array index z_1 and z_2. Instead, we assume already allocated arrays as input.

We are interested in the probability that the data of a certain set of locations has been successfully transmitted. If we additionally prove that the program is almost surely terminating for some reasonable set of fair schedulers, we can use superlinearity to prove lower bounds of probabilities for even more complex postconditions, e.g. the probability that at least half of the data have been transmitted successfully. Indeed, the program is almost surely terminating under a fairness condition. We denote the set of locations that we want to be successfully transmitted as J. For the resource invariant, we use a big separating multiplication. Its semantics is as expected: for a stack s, we connect all choices for the index variable with regular separating multiplications. The resource invariant describes the values we want to tolerate for every entry in both arrays. We join the tolerated values by a disjunction (which is the maximum in our case). We now use the resource invariant, parametrised on the set J as

$$\begin{aligned}\xi_J \quad = \quad & \left(\mathop{\bigstar}_{i \in \{0,\ldots,k\}} \max \{ [z_1 + i \mapsto 0] , [z_1 + i \mapsto 1] , [z_1 + i \mapsto 2] \} \right) \\ & \star \left(\mathop{\bigstar}_{i \in \{0,\ldots,k\} \cap J} \max \{ [z_2 + i \mapsto 0] , [z_2 + i \mapsto 1] , [z_2 + i \mapsto 2] \} \right) \\ & \star \left(\mathop{\bigstar}_{i \in \{0,\ldots,k\} \setminus J} \max \{ [z_2 + i \mapsto 0] , [z_2 + i \mapsto -1] \} \right) .\end{aligned}$$

Next, we can use the resource invariant to prove an invariant for each of the three concurrent programs. The corresponding calculations can be found in Appendix A.2. Let C_1 be the producer, C_2 the channel and C_3 the consumer. For the producer program C_1 we can prove the invariant $I_1 = 1$ with respect to the postexpectation 1, for the channel

C_2 we can prove the invariant $I_2 = [0 \leq y_2 \leq k] \cdot p^{|\{0,\ldots,y_2\} \cap J|} \cdot (1-p)^{|\{0,\ldots,y_2\} \setminus J|} + [y_2 < 0]$ with respect to the postexpectation 1, and for the consumer program C_3 we can prove the invariant $I_3 = [0 \leq y_3 \leq k] \cdot [l = |J \cap \{0,\ldots,y_3\}|] + [y_3 < 0] \cdot [l = |J|]$ with respect to the postexpectation $[l = |J|]$. Using all three invariants, we can now lower bound the probability that $l = |J|$ holds after the execution of the whole program C in Figure 8:

$$/\!\!/ \;\; [0 \leq k] \cdot p^{|\{0,\ldots,k\} \cap J|} \cdot (1-p)^{|\{0,\ldots,k\} \setminus J|} \;\; | \;\; \xi_J$$

$$l \;\texttt{:=}\; 0\,;$$

$$y_1, y_2, y_3 \;\texttt{:=}\; k\,;$$

$$/\!\!/ \;\; I_1 \star I_2 \star I_3 \;\; | \;\; \xi_J$$

$$C_1 \parallel C_2 \parallel C_3$$

$$/\!\!/ \;\; 1 \star 1 \star [l = |J|] \;\; | \;\; \xi_J$$

Here, we first use the concurrency rule to place the postexpectation $[l = |J|]$ into a separating context, thus covering all three programs. The resulting preexpectation is indeed the separating multiplication of the respective invariants. By applying the assignment rules to the first two rows, we finally get the result for a lower bound of the weakest resource-safe preexpectation with respect to resource invariant ξ_J. Thus, the lower bound $([0 \leq k] \cdot p^{|\{0,\ldots,k\} \cap J|} \cdot (1-p)^{|\{0,\ldots,k\} \setminus J|}) \star \xi_J \leq \mathsf{wlp}^S[\![C]\!]\,([l = |J|] \star \xi_J)$ also holds. We also show in Appendix A.2 how to prove the lower bound of more difficult postexpectations using superlinearity.

6 Conclusion and Future Work

Using resource invariants from Concurrent Separation Logic [53] together with quantitative reasoning from Quantitative Separation Logic [5] allows us to reason about lower-bound probabilities of realizing a postcondition. In our technique, probability mass is local to the thread. This insight gave rise to only allow qualitative expectations in the model of the environment. By this, the resource invariant only describes shared memory and lacks semantics for global probability mass.

However, we may favour a probabilistic model of the environment – for example, if the environment is a black box and only statistic information about its possible behaviours is available. More research is required for logics allowing probabilistic specifications in the environment description, especially logics allowing quantitative resource invariants. Moreover, we are only able to verify lower bounds due to the concurrent rule. We conjecture that a logic for upper bounds requires different, unknown separation connectives.

References

1 Christel Baier, Frank Ciesinski, and Markus Grosser. PROBMELA: a modeling language for communicating probabilistic processes. In *MEMOCODE*, pages 57–66. IEEE, 2004.

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

3 Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Probabilistic relational Hoare logics for computer-aided security proofs. In *MPC*, pages 1–6. Springer, 2012.

4 Kevin Batz, Ira Fesefeldt, Marvin Jansen, Joost-Pieter Katoen, Florian Keßler, Christoph Matheja, and Thomas Noll. Foundations for entailment checking in quantitative separation logic. In *ESOP*, pages 57–84. Springer, 2022.

5 Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Thomas Noll. Quantitative separation logic: a logic for reasoning about probabilistic pointer programs. *Proc. ACM Program. Lang.*, 3(POPL):34:1–34:29, 2019.

6 Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W. O'Hearn, Thomas Wies, and Hongseok Yang. Shape analysis for composite data structures. In *CAV*, pages 178–192. Springer, 2007.

7 Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, pages 115–137. Springer, 2005.

8 Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

9 Stephen Brookes and Peter W. O'Hearn. Concurrent separation logic. *ACM SIGLOG News*, 3(3):47–65, 2016.

10 Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6):26:1–26:66, 2011.

11 Michael Carbin, Sasa Misailovic, and Martin C. Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. *Commun. ACM*, 59(8):83–91, 2016.

12 Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

13 Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang. Views: Compositional reasoning for concurrent programs. In *POPL*, pages 287–300. ACM, 2013.

14 Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In *ECOOP*, pages 504–528. Springer, 2010.

15 Emanuele D'Osualdo, Julian Sutherland, Azadeh Farzan, and Philippa Gardner. TaDA live: Compositional reasoning for termination of fine-grained concurrent programs. *ACM Trans. Program. Lang. Syst.*, 43(4), 2021.

16 Ira Fesefeldt, Joost-Pieter Katoen, and Thomas Noll. Towards concurrent quantitative separation logic. *CoRR*, abs/2207.02822, 2022.

17 Keir Fraser. Practical lock-freedom. Technical Report UCAM-CL-TR-579, University of Cambridge, Computer Laboratory, 2004.

18 Ming Fu, Yong Li, Xinyu Feng, Zhong Shao, and Yu Zhang. Reasoning about optimistic concurrency using a program logic for history. In *CONCUR*, pages 388–402. Springer, 2010.

19 Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. Probabilistic programming. In *FOSE*, pages 167–181. ACM, 2014.

20 Alexey Gotsman, Josh Berdine, Byron Cook, and Mooly Sagiv. Thread-modular shape analysis. In *PLDI*, pages 266–277. ACM, 2007.

21 Friedrich Gretz, Joost-Pieter Katoen, and Annabelle McIver. Operational versus weakest pre-expectation semantics for the probabilistic guarded command language. *Performance Evaluation*, 73:110–132, 2014.

22 Sergiu Hart and Micha Sharir. Concurrent probabilistic programs, or: How to schedule if you must. *SIAM J. Comput.*, 14(4):991–1012, 1985.

23 Sergiu Hart, Micha Sharir, and Amir Pnueli. Termination of probabilistic concurrent programs. *ACM Trans. Program. Lang. Syst.*, 5(3):356–380, 1983.

24 Max P. L. Haslbeck. *Verified Quantitative Analysis of Imperative Algorithms*. PhD thesis, Technical University of Munich, Germany, 2021.

25 Mingzhang Huang, Hongfei Fu, Krishnendu Chatterjee, and Amir Kafshdar Goharshady. Modular verification for almost-sure termination of probabilistic programs. *Proc. ACM Program. Lang.*, 3(OOPSLA):129:1–129:29, 2019.

26 Samin S. Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL*, pages 14–26. ACM, 2001.

27 Kenneth E. Iverson. *A Programming Language*. John Wiley & Sons, Inc., USA, 1962.

28 Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for C and Java. In *NFM*, pages 41–55. Springer, 2011.

29 Claire Jones. *Probabilistic Non-Determinism*. PhD thesis, University of Edinburgh, 1992.

30 Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, 1983.

31 Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.*, 28:e20, 2018.

32 Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *POPL*, pages 637–650. ACM, 2015.

33 Benjamin L. Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. Weakest precondition reasoning for expected runtimes of randomized algorithms. *J. ACM*, 65(5), 2018.

34 Dexter Kozen. Semantics of probabilistic programs. In *FOCS*, pages 101–114. IEEE Computer Society, 1979.

35 Dexter Kozen. A probabilistic PDL. In *STOC*, pages 291–297. ACM, 1983.

36 Ondrej Lengál, Anthony Widjaja Lin, Rupak Majumdar, and Philipp Rümmer. Fair termination for parameterized probabilistic concurrent systems. In *TACAS*, pages 499–517. Springer, 2017.

37 Christoph Matheja. *Automated Reasoning and Randomization in Separation Logic*. PhD thesis, RWTH Aachen University, Germany, 2020.

38 Annabelle McIver and Carroll Morgan. Partial correctness for probabilistic demonic programs. *Theoretical Computer Science*, 266(1):513–541, 2001.

39 Annabelle McIver and Carroll Morgan. *Abstraction, Refinement and Proof for Probabilistic Systems*. Monographs in Computer Science. Springer, 2005.

40 Annabelle McIver, Tahiry Rabehaja, and Georg Struth. Probabilistic rely-guarantee calculus. *Theoretical Computer Science*, 655:120–134, 2016.

41 Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

42 Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In *Dependable Software Systems Engineering*, volume 50 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 104–125. IOS Press, 2017.

43 Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. Communicating state transition systems for fine-grained concurrent resources. In *ESOP*, pages 290–310. Springer, 2014.

44 Van Chan Ngo, Quentin Carbonneaux, and Jan Hoffmann. Bounded expectations: Resource analysis for probabilistic programs. *SIGPLAN Not.*, 53(4):496–512, 2018.

45 Peter W. O'Hearn. Resources, concurrency, and local reasoning. *Theoretical Computer Science*, 375(1):271–307, 2007.

46 Peter W. O'Hearn. Separation logic. *Commun. ACM*, 62(2):86–95, 2019.

47 Ruzica Piskac, Thomas Wies, and Damien Zufferey. Automating separation logic using SMT. In *CAV*, pages 773–789. Springer, 2013.

48 William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.

49 Tobias Reinhard and Bart Jacobs. Ghost signals: Verifying termination of busy waiting. In *CAV*, pages 27–50. Springer, 2021.

50 John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.

51 Joseph Tassarotti and Robert Harper. A separation logic for concurrent randomized programs. *Proc. ACM Program. Lang.*, 3(POPL):64:1–64:30, 2019.

52 Michael L. Tiomkin. Probabilistic termination versus fair termination. *Theor. Comput. Sci.*, 66(3):333–340, 1989.

53 Viktor Vafeiadis. Concurrent separation logic and operational semantics. *Electronic Notes in Theoretical Computer Science*, 276:335–351, 2011.

A Details on Examples

A.1 Additional Details on the Running Example

To recap, we are given the resource invariant $\xi = \max\{\,[r \mapsto 0],\ [r \mapsto -1]\,\}$ and have already proven:

$$
\begin{array}{l}
/\!\!/\ \ 0.5 \star 1 \ \mid\ \xi \\
\begin{array}{l||l}
 & \ /\!\!/\ \ 1 \ \mid\ \xi \\
 & \ /\!\!/\ \ 1 \star \xi \\
\ /\!\!/\ \ 0.5 \ \mid\ \xi & y \mathrel{\texttt{:=}} \langle r \rangle\,; \\
\{\,\langle r \rangle \mathrel{\texttt{:=}} 0\,\}\,[\,0.5\,]\,\{\,\langle r \rangle \mathrel{\texttt{:=}} 1\,\} & \ /\!\!/\ \ \max\{\,[y=0],\ [y=-1]\,\} \star \xi \\
\ /\!\!/\ \ 1 \ \mid\ \xi & \ /\!\!/\ \ \max\{\,[y=0],\ [y=-1]\,\} \ \mid\ \xi \\
 & \texttt{while}\,(\,y=-1\,)\,\{\,y \mathrel{\texttt{:=}} \langle r \rangle\,\}\ ; \\
 & \ /\!\!/\ \ [y=0] \ \mid\ \xi
\end{array} \\
/\!\!/\ \ 1 \star [y=0] \ \mid\ \xi
\end{array}
$$

The mutation $\langle r \rangle \mathrel{\texttt{:=}} -1$ together with the atom rule gives us the inequality

$$0.5 \star [r \mapsto -\,] \leq [r \mapsto -\,] \star (\,[r \mapsto -1] \mathrel{-\!\!\!\star} 0.5 \star \max\{\,[r \mapsto 0],\ [r \mapsto -1]\,\}\,)\ .$$

However, it is easy to verify that $([r \mapsto -1] \mathrel{-\!\!\!\star} 0.5 \star \max\{\,[r \mapsto 0],\ [r \mapsto -1]\,\})$ simplifies to 0.5, which results in the given lower bound.

For the probabilistic choice we have:

$$
\begin{array}{l}
/\!\!/\ \ 0.5 \ \mid\ \xi \\
/\!\!/\ \ 0.5 \cdot 1 + 0.5 \cdot 0 \ \mid\ \xi \\
\left\{\begin{array}{l} \ /\!\!/\ \ 1 \ \mid\ \xi \\ \langle r \rangle \mathrel{\texttt{:=}} 0 \\ \ /\!\!/\ \ 1 \ \mid\ \xi \end{array}\right\}
[\,0.5\,]
\left\{\begin{array}{l} \ /\!\!/\ \ 0 \ \mid\ \xi \\ \langle r \rangle \mathrel{\texttt{:=}} 1 \\ \ /\!\!/\ \ 1 \ \mid\ \xi \end{array}\right\} \\
/\!\!/\ \ 1 \ \mid\ \xi
\end{array}
$$

The right part is rather simple, as we can always lower bound anything by zero. The left part holds since with $[r \mapsto 0]$ the mutation is satisfied, however the value before mutating r is unknown. We can lower bound the resulting preexpectation $1 \star [r \mapsto -\,]$ by $1 \star \max\{\,[r \mapsto -1],\ [r \mapsto 0]\,\}$ and thus realise the resource invariant again. For the loop invariant $\max\{\,[y=0],\ [y=-1]\,\}$ we have:

$$
\begin{array}{l}
\ /\!\!/\ \ \max\{\,[y=0],\ [y=-1]\,\} \ \mid\ \xi \\
y \mathrel{\texttt{:=}} \langle r \rangle \\
\ /\!\!/\ \ \max\{\,[y=0],\ [y=-1]\,\} \ \mid\ \xi
\end{array}
$$

The lookup operation here results in both $[y=0]$ and $[y=1]$ to be evaluated to 1 if $[r \mapsto 0]$ and $[r \mapsto -1]$, respectively. Our resource invariant guarantees this, thus we obtain the expectation 1 and lower bound it by $\max\{\,[y=0],\ [y=-1]\,\}$. Lastly we check that it is indeed a loop invariant:

$$
\begin{aligned}
&[y=-1] \cdot \max\{\,[y=0],\ [y=-1]\,\} + [y \neq -1] \cdot [y=0] \\
={}& [y=-1] + [y=0] \\
={}& \max\{\,[y=0],\ [y=-1]\,\}
\end{aligned}
$$

A.2 Example: A Producer, a Consumer and a lossy Channel

Here we have the following program C:

```
$l$ := $0$;
$y_1, y_2, y_3$ := $k$;

while ($y_1 \geq 0$) {
    {$x_1$ := $1$} [$0.5$] {$x_1$ := $2$} ;
    <$z_1 + y_1$> := $x_1$;
    $y_1$ := $y_1 - 1$
}
||
while ($y_2 \geq 0$) {
    $x_2$ := <$z_1 + y_2$>;
    if ($x_2 \neq 0$) {
            {<$z_2 + y_2$> := $x_2$}
        [$p$]
            {<$z_2 + y_2$> := $-1$};
        $y_2$ := $y_2 - 1$
    }
}
||
while ($y_3 \geq 0$) {
    $x_3$ := <$z_2 + y_2$>;
    if ($x_3 \neq 0$) {
        if ($x_3 \neq -1$) {$l$ := $l + 1$};
        $y_3$ := $y_3 - 1$
    }
}
```

We use the resource invariant ξ_J for a set J. The set J encodes which locations in the array starting from z_2 will have an error value of -1 or a valid value of 1 or 2 after the channel inserts data into it. We use a big separating multiplication to connect all the possible instantiations using separating multiplication. That is, $\bigstar\{X\} = X$ and $\bigstar(\{X\} \cup A) = X \star \bigstar A$ for a non-empty and countable set A. ξ_J declares that all locations between z_1 and $z_1 + k$ have either value 0, 1 or 2 and all locations between z_2 and $z_2 + k$ have values 0, 1 or 2 if the offset is in J and 0 or -1 if the offset is not in J. The value 0 is always possible for all locations between z_i and $z_i + k$ since we assume 0 to be the initial value. We connect the predicates declaring possible values for the location $z_j + i$ using a maximum, which acts as a qualitative disjunction here.

$$\begin{aligned}\xi_J \quad = \quad & \left(\bigstar_{i \in \{0,\ldots,k\}} \max \{ [z_1 + i \mapsto 0], [z_1 + i \mapsto 1], [z_1 + i \mapsto 2] \} \right) \\ & \star \left(\bigstar_{i \in \{0,\ldots,k\} \cap J} \max \{ [z_2 + i \mapsto 0], [z_2 + i \mapsto 1], [z_2 + i \mapsto 2] \} \right) \\ & \star \left(\bigstar_{i \in \{0,\ldots,k\} \setminus J} \max \{ [z_2 + i \mapsto 0], [z_2 + i \mapsto -1] \} \right) .\end{aligned}$$

We will leave out computations of inequalities $X \leq Y$ for the sake of brevity and give an explanation instead. Since our representation of expectations may grow in size, we use a curly bracket after the $/\!\!/$ symbol to denote expectations which are too long for one line. Our goal is to prove a lower bound on the probability that $l = |J|$ is realised after termination. If J contains numbers outside the range between 0 and k, we may as well replace $|J|$ with $|J \cap \{0, \ldots, k\}|$. We now prove an invariant for each of the three subprograms.

For the producer C_1, we prove the invariant $I_1 = 1$ with respect to the postexpectation 1 and resource invariant ξ_J. This shows indeed that the probability of safe execution of the loop is one and that our resource invariant ξ_J almost always holds.

$$
\begin{array}{l}
\quad /\!\!/ \;\; 1 \;\mid\; \xi_J \\
\left\{ \begin{array}{l} \quad /\!\!/ \;\; 1 \;\mid\; \xi_J \\ \quad /\!\!/ \;\; [1 \in \{0, \ldots, 2\}] \;\mid\; \xi_J \\ x_1 \;\mathtt{:=}\; 1 \\ \quad /\!\!/ \;\; [x_1 \in \{0, \ldots, 2\}] \;\mid\; \xi_J \end{array} \right\} [\,0.5\,] \left\{ \begin{array}{l} \quad /\!\!/ \;\; 1 \;\mid\; \xi_J \\ \quad /\!\!/ \;\; [2 \in \{0, \ldots, 2\}] \;\mid\; \xi_J \\ x_1 \;\mathtt{:=}\; 2 \\ \quad /\!\!/ \;\; [x_1 \in \{0, \ldots, 2\}] \;\mid\; \xi_J \end{array} \right\} ; \\
\quad /\!\!/ \;\; [x_1 \in \{0, \ldots, 2\}] \;\mid\; \xi_J \\
\mathtt{<}\, z_1 + y_1 \,\mathtt{>} \;\mathtt{:=}\; x_1 \,; \\
\quad /\!\!/ \;\; 1 \;\mid\; \xi_J \\
y_1 \;\mathtt{:=}\; y_1 - 1 \\
\quad /\!\!/ \;\; 1 \;\mid\; \xi_J
\end{array}
$$

The inequality

$$[x_1 \in \{0, \ldots, 2\}] \star \xi_J \leq [z_1 + y_1 \mapsto -] \star ([z_1 + y_1 \mapsto x_1] \mathrel{-\!\!\star} (1 \star \xi_J))$$

resulting from the mutation $\mathtt{<}\, z_1 + y_2 \,\mathtt{>} \;\mathtt{:=}\; x_1$ together with the atom rule holds because for $[z_1 + y_1 \mapsto x_1] \mathrel{-\!\!\star} (1 \star \xi_J)$ to be non-zero, x_1 must coincide with ξ_J – thus x_1 has to be either 0, 1 or 2. Furthermore, we have that $[z_1 + y_1 \mapsto i] \leq [z_1 + y_1 \mapsto -]$ holds for every i, and obtain by this that $i \in \{0, \ldots, 2\}$, with which we re-establish ξ_J. We have $[y_1 \geq 0] \cdot 1 + [y_1 < 0] \cdot 1 = 1$ and therefore 1 is a loop invariant.

For the channel C_2, we use the shorthand notation $P(y)$ to denote cumulated probability mass and define it as

$$P(y) \quad = \quad p^{|\{0, \ldots, y\} \cap J|} \cdot (1 - p)^{|\{0, \ldots, y\} \setminus J|} \;.$$

This shorthand notation gives us the probability that all data with offset 0 up to y are transferred according to J. That is, if an element should have been transferred successfully, we multiply with p and if not with $1 - p$ for every location up to y_2. We prove for the invariant $I_2 = [0 \leq y_2 \leq k] \cdot P(y_2) + [y_2 < 0]$ with respect to the postexpectation 1 and resource invariant ξ_J:

$$
\begin{array}{l}
\quad /\!\!/ \;\; [0 \leq y_2 \leq k] \cdot P(y_2) + [y_2 < 0] \;\mid\; \xi_J \\
\quad /\!\!/ \;\; \left\{ \begin{array}{l} [x_2 \neq 0] \cdot [0 \leq y_2 \leq k] \cdot P(y_2) \\ + [x_2 = 0] \cdot ([0 \leq y_2 \leq k] \cdot P(y_2) + [y_2 < 0]) \end{array} \right. \;\left|\; \xi_J \right. \\
x_2 \;\mathtt{:=}\; \mathtt{<}\, z_1 + y_2 \,\mathtt{>} \,; \\
\quad /\!\!/ \;\; \left\{ \begin{array}{l} [x_2 \neq 0] \cdot [0 \leq y_2 \leq k] \cdot P(y_2 - 1) \cdot (p \cdot [y_2 \in J] \cdot [x_2 \in \{1, \ldots, 2\}] \\ \qquad + (1 - p) \cdot [y_2 \notin J]) \\ + [x_2 = 0] \cdot ([0 \leq y_2 \leq k] \cdot P(y_2) + [y_2 < 0]) \end{array} \right. \;\left|\; \xi_J \right. \\
\mathtt{if} \; (\, x_2 \neq 0 \,) \; \{ \\
\qquad /\!\!/ \;\; [0 \leq y_2 \leq k] \cdot P(y_2 - 1) \cdot (p \cdot [y_2 \in J] \cdot [x_2 \in \{0, \ldots, 2\}] + (1 - p) \cdot [y_2 \notin J]) \;\mid\; \xi_J \\
\qquad \{ \\
\qquad\quad /\!\!/ \;\; ([0 \leq y_2 \leq k] \cdot P(y_2 - 1)) \cdot [y_2 \in J] \cdot [x_2 \in \{0, \ldots, 2\}] \;\mid\; \xi_J
\end{array}
$$

```
          ⫽ $([1 \le y_2 \le k] \cdot P(y_2 - 1) + [y_2 = 0]) \cdot [y_2 \in J] \cdot [x_2 \in \{0, \ldots, 2\}] \quad | \quad \xi_J$
        < $z_2 + y_2$ > := $x_2$
          ⫽ $[1 \le y_2 \le k + 1] \cdot P(y_2 - 1) + [y_2 < 1] \quad | \quad \xi_J$
        }
      [ $p$ ]
        {
          ⫽ $([0 \le y_2 \le k] \cdot P(y_2 - 1)) \cdot [y_2 \notin J] \quad | \quad \xi_J$
          ⫽ $([1 \le y_2 \le k] \cdot P(y_2 - 1) + [y_2 = 0]) \cdot [y_2 \notin J] \quad | \quad \xi_J$
        < $z_2 + y_2$ > := $-1$ ;
          ⫽ $[1 \le y_2 \le k + 1] \cdot P(y_2 - 1) + [y_2 < 1] \quad | \quad \xi_J$
        }
      ⫽ $[1 \le y_2 \le k + 1] \cdot P(y_2 - 1) + [y_2 < 1] \quad | \quad \xi_J$
      ⫽ $[0 \le y_2 - 1 \le k] \cdot P(y_2 - 1) + [y_2 - 1 < 0] \quad | \quad \xi_J$
    $y_2$ := $y_2 - 1$
      ⫽ $[0 \le y_2 \le k] \cdot P(y_2) + [y_2 < 0] \quad | \quad \xi_J$
  }
    ⫽ $[0 \le y_2 \le k] \cdot P(y_2) + [y_2 < 0] \quad | \quad \xi_J$
```

We explain some of the difficult inequalities in the previous proof. We start with the inequality

$$\begin{aligned} &(([1 \le y_2 \le k] \cdot P(y_2 - 1) + [y_2 = 0]) \cdot [y_2 \notin J]) \star \xi_J \\ \le\ &[z_2 + y_2 \mapsto -\,] \star ([z_2 + y_2 \mapsto -1] \mathbin{-\!\!\star} ([1 \le y_2 \le k+1] \cdot P(y_2 - 1) + [y_2 < 1]) \star \xi_J \end{aligned}$$

resulting from the mutation < $z_2 + y_2$ > := -1 together with the atom rule. This inequality holds since for the part $[z_2 + y_2 \mapsto -1] \mathbin{-\!\!\star} \ldots$ to be non-zero, we require $y_2 \notin J$ due to ξ_J. We lower bound all evaluations where $y_2 < 0$ by 0 as we can not infer any information about these locations from ξ_J. Afterwards, we can lower bound $[z_2 + y_2 \mapsto -\,]$ by $[z_2 + y_2 \mapsto i]$ for every i and thus re-establish ξ_J.

Next we have the inequality

$$\begin{aligned} &(([1 \le y_2 \le k] \cdot P(y_2 - 1) + [y_2 = 0]) \cdot [y_2 \in J] \cdot [x_2 \in \{0, \ldots, 2\}]) \star \xi_J \\ \le\ &[z_2 + y_2 \mapsto -\,] \star ([z_2 + y_2 \mapsto x_2] \mathbin{-\!\!\star} ([1 \le y_2 \le k+1] \cdot P(y_2 - 1) + [y_2 < 1]) \star \xi_J \end{aligned}$$

resulting from the mutation < $z_2 + y_2$ > := x_2 together with the atom rule. Here we assume that the location y_2 is in J and obtain that $x_2 \in \{0, \ldots, 2\}$. We lower bound any outcome of y_2 not in J by 0 because we already know that we will eventually set the term to 0 due to the previous lookup. Next we establish ξ_J back from lower bounding $[z_2 + y_2 \mapsto -\,]$.

We have the inequality

$$\begin{aligned} &[x_2 \neq 0] \cdot [0 \le y_2 \le k] \cdot P(y_2) + [x_2 = 0] \cdot ([0 \le y_2 \le k] \cdot P(y_2) + [y_2 < 0]) \star \xi \\ \le &\sup_{v \in \mathbb{Z}} [z_1 + y_2 \mapsto v] \star ([z_1 + y_2 \mapsto v] \mathbin{-\!\!\star} \\ &([v \neq 0] \cdot [0 \le y_2 \le k] \cdot P(y_2 - 1) \cdot (p \cdot [y_2 \in J] \cdot [v \in \{1, \ldots, 2\}] + (1 - p) \cdot [y_2 \notin J]) \\ &+ [v = 0] \cdot ([0 \le y_2 \le k] \cdot P(y_2) + [y_2 < 0])) \star \xi) \end{aligned}$$

resulting from the lookup x_2 `:=` `<` $z_1 + y_2$ `>` together with the atom rule. We will consider both cases separately. Let us assume that v is not 0. Then either y_2 is in J and v is either 1 or 2 to make $p \cdot [y_2 \in J] \cdot [v \in \{1, \ldots, 2\}]$ not zero, or y_2 is not in J. Then, however, v needs to be -1, because else ξ_J will evaluate to zero. Both cases can then be used to turn $P(y_2 - 1)$ into $P(y_2)$. In both cases, we can also use $[z_1 + y_2 \mapsto v]$ to re-establish the resource invariant ξ_J. If, on the other side, v is 0, we do not get any new information, but also do not need to update $P(y_2)$, and directly re-establish the resource invariant ξ_J.

Due to

$$\begin{aligned} & [y_2 \geq 0] \cdot ([0 \leq y_2 \leq k] \cdot P(y_2) + [y_2 < 0]) + [y_2 < 0] \cdot 1 \\ = \; & [0 \leq y_2 \leq k] \cdot P(y_2) + [y_2 < 0] \end{aligned}$$

we establish the loop invariant with respect to postexpectation 1.

For the consumer C_3 we require a loop invariant that checks if l indeed matches the size of the set J. We prove the loop invariant $I_3 = [0 \leq y_3 \leq k] \cdot [y_3 + l = |J \cap \{0, \ldots, y_3\}|]$ with respect to the postexpectation $[l = |J|]$ and the resource invariant ξ_J:

```
// $[0 \leq y_3 \leq k] \cdot [l = |J \cap \{y_3 + 1, \ldots, k\}|] \quad | \quad \xi_J$
$x_3$ := < $z_2 + y_3$ > ;
// $\left\{ \begin{array}{l} \quad [x_3 = -1] \cdot [1 \leq y_3 \leq k+1] \cdot [l = |J \cap \{y_3, \ldots, k\}|] \\ + [x_3 = -1] \cdot [y_3 < 1] \cdot [l = |J|] \\ + [x_3 \neq 0] \cdot [x_3 \neq -1] \cdot [1 \leq y_3 \leq k+1] \cdot [l + 1 = |J \cap \{y_3, \ldots, k\}|]) \\ + [x_3 \neq 0] \cdot [x_3 \neq -1] \cdot [y_3 < 1] \cdot [l + 1 = |J|] \\ + [x_3 = 0] \cdot [0 \leq y_3 \leq k] \cdot [l = |J \cap \{y_3 + 1, \ldots, k\}|] \\ + [x_3 = 0] \cdot [y_3 < 0] \cdot [l = |J|] \end{array} \right. \quad \Bigg| \quad \xi_J$
if ( $x_3 \neq 0$ ) {
   // $\left\{ \begin{array}{l} \quad [x_3 = -1] \cdot [1 \leq y_3 \leq k+1] \cdot [l = |J \cap \{y_3, \ldots, k\}|] \\ + [x_3 = -1] \cdot [y_3 < 1] \cdot [l = |J|] \\ + [x_3 \neq -1] \cdot [1 \leq y_3 \leq k+1] \cdot [l + 1 = |J \cap \{y_3, \ldots, k\}|] \\ + [x_3 \neq -1] \cdot [y_3 < 1] \cdot [l + 1 = |J|] \end{array} \right. \quad \Bigg| \quad \xi_J$
  if ( $x_3 \neq -1$ ) {
     // $[1 \leq y_3 \leq k+1] \cdot [l + 1 = |J \cap \{y_3, \ldots, k\}|] + [y_3 < 1] \cdot [l + 1 = |J|] \quad | \quad \xi_J$
    $l$ := $l + 1$
     // $[1 \leq y_3 \leq k+1] \cdot [l = |J \cap \{y_3, \ldots, k\}|] + [y_3 < 1] \cdot [l = |J|] \quad | \quad \xi_J$
  } ;
   // $[1 \leq y_3 \leq k+1] \cdot [l = |J \cap \{y_3, \ldots, k\}|] + [y_3 < 1] \cdot [l = |J|] \quad | \quad \xi_J$
   // $[0 \leq y_3 - 1 \leq k] \cdot [l = |J \cap \{y_3, \ldots, k\}|] + [y_3 - 1 < 0] \cdot [l = |J|] \quad | \quad \xi_J$
  $y_3$ := $y_3 - 1$
   // $[0 \leq y_3 \leq k] \cdot [l = |J \cap \{y_3 + 1, \ldots, k\}|] + [y_3 < 0] \cdot [l = |J|] \quad | \quad \xi_J$
}
// $[0 \leq y_3 \leq k] \cdot [l = |J \cap \{y_3 + 1, \ldots, k\}|] + [y_3 < 0] \cdot [l = |J|] \quad | \quad \xi_J$
```

Here we will take a closer look at the inequality

$$\begin{aligned}
&([1 \leq y_3 \leq k] \cdot [l = |J \cap \{y_3 + 1, \ldots, k\}|] + [y_3 = 0] \cdot [l = |J \cap \{1, \ldots, k\}|]) \star \xi_J \\
\leq\ & \sup_{v \in \mathbb{Z}} [z_2 + y_2 \mapsto v] \star ([z_2 + y_2 \mapsto v] \mathrel{-\!\!\star} \\
&\quad ([v = -1] \cdot [1 \leq y_3 \leq k + 1] \cdot [l = |J \cap \{y_3, \ldots, k\}|] \\
&\quad + [v = -1] \cdot [y_3 < 1] \cdot [l = |J|] \\
&\quad + [v \neq 0] \cdot [v \neq -1] \cdot [1 \leq y_3 \leq k + 1] \cdot [l + 1 = |J \cap \{y_3, \ldots, k\}|]) \\
&\quad + [v \neq 0] \cdot [v \neq -1] \cdot [y_3 < 1] \cdot [l + 1 = |J|] \\
&\quad + [v = 0] \cdot [0 \leq y_3 \leq k] \cdot [l = |J \cap \{y_3 + 1, \ldots, k\}|] \\
&\quad + [v = 0] \cdot [y_3 < 0] \cdot [l = |J|]) \star \xi_J)
\end{aligned}$$

due to the lookup x_3 `:=` `<`$z_2 + y_2$`>` together with the atom rule. We consider all cases separately.

- First, let v be -1
  - If moreover y_3 is between 1 and $k + 1$, then we can directly lower bound the case that y_3 is $k + 1$ by zero as ξ_J does not have carry information for this location. Because v is -1, we know that y_3 is not in J due to ξ_J. Thus, we also have that $|J \cap \{y_3 + 1, \ldots, k\}| = |J \cap \{y_3, \ldots, k\}|$.
  - If y_3 is below 1, the same reasoning holds, with the difference that we lower bound the expectation for every value of y_3 below 0 as zero and consider only the case where y_3 is 0.
- In the case that v is neither 0 nor -1, we first observe that only 1 and 2 are valid values, because ξ_J does not allow any other value for y_3 between 0 and k.
- In the cases where v is either $k + 1$ or below 0, we just lower bound the formula by zero. However, for the latter cases we have $|J \cap \{y_3 + 1, \ldots, k\}| + 1 = |J \cap \{y_3, \ldots, k\}|$.
- Lastly, in the case that v is 0, the expression already matches the target lower bound, but again, we lower bound the formula by zero if y_3 has a value below 0.

Moreover, we have

$$\begin{aligned}
&[y_3 \geq 0] \cdot ([0 \leq y_3 \leq k] \cdot [l = |J \cap \{y_3 + 1, \ldots, k\}|] + [y_3 < 0] \cdot [l = |J|]) \\
=\ & [0 \leq y_3 \leq k] \cdot [l = |J \cap \{y_3 + 1, \ldots, k|\}] + [y_3 < 0] \cdot [l = |J|]
\end{aligned}$$

and thus established a loop invariant with respect to postexpectation $[l = |J|]$.

Now we can combine all three results

$/\!\!/\ \ P(k) \cdot [0 \leq k] \ \ | \ \ \xi_J$

$/\!\!/\ \ [0 \leq k] \cdot P(k) + [k < 0] \cdot [0 = |J|] \ \ | \ \ \xi_J$

l `:=` 0 ;

$/\!\!/\ \ [0 \leq k] \cdot [l = 0] \cdot P(k) + [k < 0] \cdot [l = |J|] \ \ | \ \ \xi_J$

$$/\!\!/ \left\{ \begin{array}{l} 1 \\ \star\ ([0 \leq k] \cdot P(k) + [k < 0]) \\ \star\ ([0 \leq k] \cdot [l = |J \cap \{k + 1, \ldots, k\}|] + [k < 0] \cdot [l = |J|]) \end{array} \right| \ \ \xi_J$$

y_1, y_2, y_3 `:=` k ;

$$/\!\!/ \left\{ \begin{array}{l} 1 \\ \star\ ([0 \leq y_2 \leq k] \cdot P(y_2) + [y_2 < 0]) \\ \star\ ([0 \leq y_3 \leq k] \cdot [l = |J \cap \{y_3 + 1, \ldots, k\}|] + [y_3 < 0] \cdot [l = |J|]) \end{array} \right| \ \ \xi_J$$

$$\begin{aligned}&C_1 \parallel C_2 \parallel C_3\\ &/\!\!/\;\; 1 \star 1 \star [l = |J|] \;\;|\;\; \xi_J\end{aligned}$$

and we have for the whole program C and a set of schedulers $S \subseteq \mathbb{S}$:

$$\begin{aligned}
&(P(k) \cdot [0 \leq k]) \leq \mathsf{wrlp}[\![C]\!]\,([l = |J|] \mid \xi_J) \\
\text{implies } &(P(k) \cdot [0 \leq k]) \star \xi_J \leq \mathsf{wrlp}[\![C]\!]\,([l = |J|] \star \xi_J \mid [\mathsf{emp}]) && \text{(share)}\\
\text{implies } &(P(k) \cdot [0 \leq k]) \star \xi_J \leq \mathsf{wlp}^S[\![C]\!]\,([l = |J|] \star \xi_J) && (\mathsf{wlp\text{-}wrlp})
\end{aligned}$$

We can use this to prove the lower bound of probabilities for even more elaborated postconditions if we have a set of schedulers $S \subseteq \mathbb{S}$ such that C is almost surely terminating with respect to S. One of these is the probability that at least half of the messages are sent successfully, i.e., the probability of the postexpectation $\left[k+1 \geq l \geq \frac{k+1}{2}\right]$ – or equivalently $\sum_{\frac{k+1}{2} \leq j \leq k+1} [l = j]$. For this, we use the resource invariant $\xi_j = \max_{J \subseteq \{0,\ldots,k\}, |J| = j} \xi_J$ where ξ_J is defined as previous. Although we call ξ_j a resource invariant, we never prove that it is a resource invariant. We only prove that ξ_J is a resource invariant. We can now compute:

$$\begin{aligned}
&\mathsf{wlp}^S[\![C]\!]\,([l = |J|] \star \xi_J) \geq (P(k) \cdot [0 \leq k]) \star \xi_J \\
\text{implies } &\mathsf{wlp}^S[\![C]\!]\left(\max_{J \subseteq \{0,\ldots,k\}, |J| = j} [l = |J|] \star \xi_J\right) \geq \max_{J \subseteq \{0,\ldots,k\}, |J| = j} (P(k) \cdot [0 \leq k]) \star \xi_J \\
& && \text{(max)}\\
\text{implies } &\mathsf{wlp}^S[\![C]\!]\,([l = j] \star \xi_j) \geq (p^j \cdot (1-p)^{k-j+1} \cdot [0 \leq k]) \star \xi_j && \text{(Definition of } \xi_j)\\
\text{implies } &\mathsf{wlp}^S[\![C]\!]\left(\sum_{\frac{k+1}{2} \leq j \leq k+1} [l = j] \star \xi_j\right) \geq \sum_{\frac{k+1}{2} \leq j \leq k+1} (p^j \cdot (1-p)^{k-j+1} \cdot [0 \leq k]) \star \xi_j \\
& && \text{(Superlinearity)}\\
\text{implies } &\mathsf{wlp}^S[\![C]\!]\left(\left[k+1 \geq l \geq \frac{k+1}{2}\right] \star \xi_j\right) \\
&\quad \geq \left(\sum_{\frac{k+1}{2} \leq j \leq k+1} p^j \cdot (1-p)^{k-j+1} \cdot [0 \leq k]\right) \star \xi_j \\
& && (\xi_j \text{ is precise and } \left[k+1 \geq l \geq \tfrac{k+1}{2}\right] \text{ as above})
\end{aligned}$$

We could drop the resource invariant ξ_j inside wlp^S due to monotonicity of wlp^S, for which we do not provide a proof. However, this shows that we can use superlinearity to partition a big problem in smaller problems and afterwards reason about these smaller problems with the help of easier resource invariants, as it is standard in probability theory.