Modular Reasoning about Error Bounds for Concurrent Probabilistic Programs

KWING HEI LI, Aarhus University, Denmark
ALEJANDRO AGUIRRE, Aarhus University, Denmark
SIMON ODDERSHEDE GREGERSEN, New York University, USA
PHILIPP G. HASELWARTER, Aarhus University, Denmark
JOSEPH TASSAROTTI, New York University, USA
LARS BIRKEDAL, Aarhus University, Denmark

We present Coneris, the first *higher-order concurrent separation logic* for reasoning about error probability bounds of higher-order concurrent probabilistic programs with higher-order state. To support modular reasoning about concurrent (non-probabilistic) program modules, state-of-the-art program logics internalize the classic notion of linearizability within the logic through the concept of *logical atomicity*. In Coneris, we extend this idea to probabilistic concurrent program modules by capturing a novel notion of *randomized logical atomicity* within the logic. To do so, Coneris utilizes *presampling tapes* and a novel *probabilistic update modality* to describe how state is changed probabilistically at linearization points. We demonstrate this approach by means of smaller synthetic examples and larger case studies. All of the presented results, including the meta-theory, have been mechanized in the Rocq prover and the Iris separation logic framework.

1 Introduction

Probabilistic data structures, such as approximate counters, skip lists, or Bloom filters are widely used in concurrent programming. These data structures can improve time and space efficiency compared to their deterministic counterparts. However, some probabilistic data structures may return wrong results with a small probability. Analyzing and ensuring this probability of error is sufficiently small is essential for using these data structures. But this analysis is challenging because probabilistic programs often have unintuitive behaviors, which are only made more complicated when probabilistic behaviors are combined with concurrency. Because randomness and concurrency both introduce non-determinism, any analysis must take into account the large range of possible outcomes that can arise from this non-determinism.

For just concurrency alone without randomness, a number of verification techniques have been developed that abstract away from concurrent non-determinism. For example, Concurrent Separation Logic (CSL) [33] allows for threads in a concurrent program to be verified in a *local* way, without having to consider the effects of interference from other threads at every step of execution. A key feature of modern concurrent separation logics is support for proving that data structures are *logically atomic* [15, 23, 27, 34]. Logical atomicity allows clients to reason *as if* a concurrent data structure had a logical state that is updated atomically at a single point in time during each operation. This internalizes the idea of *linearizability*, a standard notion of correctness for concurrent data structures, and enables modular and compositional proofs.

It is natural to consider whether similar techniques can be applied to reason about *randomized* concurrent data structures. Prior work has explored extending concurrent separation logic to the randomized setting [17, 35, 38]. However, none of these prior logics support compositional reasoning about data structures because they lack support for reasoning about logical atomicity. Indeed, even developing a suitable notion of what logical atomicity would look like in the presence of

Authors' Contact Information: Kwing Hei Li, Aarhus University, Denmark, hei.li@cs.au.dk; Alejandro Aguirre, Aarhus University, Denmark, alejandro@cs.au.dk; Simon Oddershede Gregersen, New York University, USA, s.gregersen@nyu.edu; Philipp G. Haselwarter, Aarhus University, Denmark, pgh@cs.au.dk; Joseph Tassarotti, New York University, USA, jt4767@cs.nyu.edu; Lars Birkedal, Aarhus University, Denmark, birkedal@cs.au.dk.

randomization is challenging. Whereas prior efforts developing non-randomized logical atomicitiy could draw inspiration and intuition from the notion of linearizability, there is no widely-accepted analogue of linearizability that is suitable for randomized data structures. Indeed, prior work has shown ways in which standard notions of linearizability do not suffice when clients of a data structure can make randomized choices [18].

In this paper, we develop an appropriate notion of *randomized logical atomicity* in the context of Coneris, a new concurrent separation logic. Coneris is a *probabilistic higher-order concurrent separation logic* for reasoning about error bounds of ConcRandML programs, a ML-like language with support for discrete random sampling, unstructured concurrency, higher-order functions, and higher-order dynamically-allocated local state.

On the surface level, Coneris retains all the standard rules of higher-order separation logic for concurrent programs, along with modern features like expressive impredicative invariants and custom ghost resources. From the probabilistic side, Coneris inherits two kinds of separation logic resources from previous logics for reasoning about probabilities, specifically *presampling tapes* first introduced in Clutch [20], and *error credits* first introduced in Eris [3]. The former are used to reason about random choices that a program will take in the future, while the latter are used to track an upper bound on the probability of some error occurring during execution.

Using its higher-order features, Coneris encodes randomized logical atomicity by adapting the earlier HOCAP [34] approach to logical atomicity. To do so, we introduce a novel *probabilistic update modality*, written as $\leadsto P$. Intuitively, the modality is used to describe a probabilistic logical update to a piece of ghost state of the program. In particular, we can use it to reason as if the randomness that the program uses is atomically drawn at one single point in time, which is crucial in writing and proving modular specifications.

We demonstrate the flexibility of our approach by verifying a selection of data structures. For example, we verify the correctness of a concurrent hash function. The hash module is subsequently used in an efficient concurrent implementation of a Bloom filter, and we derive a strict error bound for a client program that uses the Bloom filter. These examples utilize rich language features such as higher-order functions and local state, and display non-trivial interactions between concurrency and probability. As far as we are aware, even verifying the simpler examples are out-of-scope for previous techniques, let alone reasoning about them *modularly*.

Contributions. In summary, we make the following contributions:

- The first concurrent and probabilistic higher-order separation logic for reasoning about error bounds of programs written in ConcRandML, a probabilistic concurrent higher-order programming language with higher-order references.
- A novel probabilistic update modality that we use to capture a probabilistic notion of logical atomicity.
- An extension of the HOCAP approach to the probabilistic setting that allows us to write and prove modular specifications of randomized concurrent data structures.
- A selection of case studies showcasing our approach to modular verification of higher-order concurrent probabilistic data structures.
- Full mechanization of all results in the Rocq prover [36], using the Iris separation logic framework [25] and the Coquelicot real analysis library [13].

Outline. In §2, we demonstrate how Coneris is used to reason about programs that feature both concurrency and probability, we and highlight some of the key challenges that arise from this combination. We then present the syntax and semantics of our language ConcRandML in §3. In §4, we present a collection of program logic rules and we show how to apply them to reason about a

concurrent randomized counter module in §5. Subsequently in §6, we showcase Coneris on a range of concurrent probabilistic data structure examples. Finally, we discuss related work and conclude with ideas for future work in §8 and §9, respectively.

2 Motivation and Technical Challenges

We first recall the features of the Eris logic [3] for reasoning about error bounds of sequential programs, then we discuss how Coneris extends these ideas to the concurrent setting, illustrating some of the challenges that arise when reasoning about error bounds in the presence of concurrency.

Sequential Error Reasoning with Error Credits. Eris is a separation logic that introduces a new assertion called *error credits* written $↯(\varepsilon)$, where $0 \le \varepsilon \le 1$ is a real number. This assertion represents a budget upper bounding the probability that a specification can fail to hold. In particular, an Eris Hoare triple of the form $\{P * ↯(\varepsilon)\}\ e\ \{x.\ Q\}$ implies that if we execute e in a state satisfying P, then the probability that e crashes or returns a value x that violates Q is at most ε.

To illustrate how the $↯(\varepsilon)$ assertion is used, consider the following program as a simple example:

$$twoAdd \triangleq \mathsf{let}\ l = \mathsf{ref}\ 0\ \mathsf{in}\ l \leftarrow (!\, l + \mathsf{rand}\ 3); l \leftarrow (!\, l + \mathsf{rand}\ 3); !\, l$$

Here rand N is a probabilistic construct that samples a value uniformly from $\{0, \ldots, N\}$. In particular, rand 3 returns a random number from the set $\{0, \ldots, 3\}$ with probability $1/4$ each. The *twoAdd* program first allocates a reference l initialized to 0, then adds the result of a probabilistic sampling rand 3 to the value in l twice, and concludes by reading the number in l.

Suppose we want to prove an upper bound on the probability that the program returns 0. This happens only if both calls to rand 3 return 0, which occurs with probability at most $1/4 \cdot 1/4 = 1/16$. We can capture this as a specification in Eris by proving $\{↯(1/16)\}\ twoAdd\ \{x.\ x > 0\}$.

Eris provides 3 key rules for working with error credits:[1]

$$\textsc{err-split}\ \frac{↯(\varepsilon_1 + \varepsilon_2)}{↯(\varepsilon_1) * ↯(\varepsilon_2)} \qquad \textsc{ht-rand-exp}\ \frac{\mathbb{E}_{\mathfrak{U}N}[\mathcal{F}] \le \varepsilon}{\{↯(\varepsilon)\}\ \mathsf{rand}\ N\ \{n.\ ↯(\mathcal{F}(n)) * n \in \{0..N\}\}} \qquad \textsc{err-1}\ \frac{↯(1)}{\mathsf{False}}$$

The first rule says that error credits can be split and joined together. The second rule says that when the program makes a randomized choice, we can re-distribute error credits along different branches of the randomized outcome, so long as the *expected value* or *average* amount of error credit does not increase. Finally, the third rule says that an error credit of 1 implies False, *i.e.*, we can deduce anything, which intuitively follows from the idea that an upper bound of probability 1 is trivial. Figure 1 shows a proof outline using these rules to derive the Hoare triple stated above for *twoAdd*. The key steps in this proof are to apply the HT-RAND-EXP rule twice to reason about the two rand 3 statements in the proof. The first time the rule is applied, we start with $↯(1/16)$, and instantiate $\mathcal{F}$ in the rule to be the function $\mathcal{F}(n) \triangleq \frac{1}{4} \cdot [n = 0]$ where $[P]$ evaluates to 1 if $P(a)$ is true and to 0 otherwise. That is, we end up with $↯(1/4)$ in the case where rand 3 returned 0, and $↯(0)$ otherwise. For the latter cases, the remainder of the proof is trivial, since the value in l is already greater than 0. For the former case, on the next call to rand 3, we again apply HT-RAND-EXP, setting $\mathcal{F}'(m) \triangleq [n = 0 \wedge m = 0]$. Thus, when rand 3 again returns 0, we end up with $↯(1)$. In this case, l still contains 0, but we can use ERR-1 to finish the proof. For the other cases, l will be greater than 0, so the postcondition follows directly.

[1] We write inference rules with a double horizontal line to mean the rule can be applied in either direction.

$\{↯(\frac{1}{16})\}$
  let l = ref 0 in
$\{l \mapsto 0 * ↯(\frac{1}{16})\}$
  $l \leftarrow (!\, l + \text{rand}\ 3)$; (apply HT-RAND-EXP using $\mathcal{F}(x) \triangleq \frac{1}{4} \cdot [x = 0]$)
$\{\exists n.\, l \mapsto n * \big((n = 0 \land ↯(\frac{1}{4})) \lor n \neq 0\big)\}$
  $l \leftarrow (!\, l + \text{rand}\ 3)$; (apply HT-RAND-EXP using $\mathcal{F}'(x) \triangleq [n = 0 \land x = 0]$)
$\{\exists m.\, l \mapsto m * ((m = 0 \land ↯(1)) \lor m \neq 0)\}$
$\{\exists m.\, l \mapsto m * m \neq 0\}$ (discharge case $m = 0$ using ERR-1 from $↯(1)$)
  $!\, l$
$\{x.\, x > 0\}$

Fig. 1. Proof outline for the Hoare triple $\{↯(1/16)\}$ *twoAdd* $\{x.\, x > 0\}$.

Concurrent Error Bounds in Coneris. Coneris generalizes Eris's error credit reasoning to the concurrent setting. To illustrate some of the key ideas and challenges in doing so, consider the following concurrent variation of the *twoAdd* example:

$$\mathit{conTwoAdd} \triangleq \text{let}\ l = \text{ref}\ 0\ \text{in}\ (\text{faa}\ l\ (\text{rand}\ 3)\ |||\ \text{faa}\ l\ (\text{rand}\ 3))\,;!\, l$$

Here $|||$ represents parallel composition of two threads, and faa l n is a fetch-and-add command that atomically adds n to the value stored in l and returns the value prior to the addition. Intuitively, no matter which order the faa commands execute in the two threads, the probability that the final $!\, l$ at the end of the program returns 0 is again bounded above by $1/16$.

Coneris allows us to show this by proving a Hoare triple of a similar form as the one we saw for *twoAdd*. More precisely, in Coneris, the intuitive meaning of a Hoare triple $\{P * ↯(\varepsilon)\}\ e\ \{Q\}$ is that, "*for all possible schedulings of threads*, if P holds, then the probability that e reaches an error state or returns a value that violates Q is at most ε".

All of Eris's proof rules for error credits also hold in Coneris. To reason about parallel execution, Coneris additionally has the familiar parallel composition rule from concurrent separation logic:

$$\frac{\{P_1\}\ e_1\ \{v_1.\, Q_1\ v_1\} \qquad \{P_2\}\ e_2\ \{v_2.\, Q_2\ v_2\}}{\{P_1 * P_2\}\ e_1\ |||\ e_2\ \{(v_1, v_2).\, Q_1\ v_1 * Q_2\ v_2\}}\ \text{HT-PAR-COMP}$$

To apply this rule we have to divide up the precondition into two separate parts, P_1 and P_2, and show that they suffice as preconditions for the two threads e_1 and e_2, respectively. We already saw with ERR-SPLIT that we can split error credits, so for the *conTwoAdd* example we might try to split the initial error budget of $↯(1/16)$ in half, giving each thread $↯(1/32)$. However, we would soon run into two issues:

(1) As in the sequential case, we want to apply HT-RAND-EXP to try to distribute all of the credits to the cases where the rand 3 commands return 0. However, if each thread has $↯(1/32)$ and applies HT-RAND-EXP to reason about the rand 3 it executes, then after applying the rule, it can have at most $↯(1/8)$ for the case where the rand 3 returns 0. Combining two $↯(1/8)$ in the post-condition of the parallel composition rule, we would end up with a total of $↯(2/8)$ for the case where both threads add 0 to the counter. But this is not enough to apply ERR-1, and so we would not be able to prove the intended postcondition.
(2) Both threads need to modify the shared location l, so they both need to have "ownership" of the points-to assertion for l. However, $l \mapsto 0 \not\vdash l \mapsto 0 * l \mapsto 0$, so we cannot pass ownership of this assertion in the precondition of both threads when applying HT-PAR-COMP.

The second problem has a well-known solution in CSL, namely *invariant assertions*. Fortunately, as we will see, it turns out that invariants also provide a solution to the first problem.

Coneris provides Iris-style invariant assertions of the form $\boxed{I}$, which states that an assertion I is an invariant of program execution. These assertions support the following rules:

$$\textsc{ht-inv-alloc}\ \frac{\{\boxed{I} * P\}\ e\ \{Q\}}{\{I * P\}\ e\ \{Q\}} \qquad \textsc{inv-dup}\ \frac{\boxed{I}}{\boxed{I} * \boxed{I}} \qquad \textsc{ht-inv-open}\ \frac{e\ \text{atomic} \qquad \{I * P\}\ e\ \{I * Q\}}{\{\boxed{I} * P\}\ e\ \{\boxed{I} * Q\}}$$

The first rule HT-INV-ALLOC says that we can allocate an invariant assertion $\boxed{I}$ if we know that I holds in the precondition. Invariant assertions are *duplicable*, meaning that we can produce multiple copies using INV-DUP, so that when applying HT-PAR-COMP, each thread can have a copy of $\boxed{I}$ in its precondition. Finally, threads can access the assertion I inside of the invariant using HT-INV-OPEN, which requires that the invariant is restablished after the atomic expression e finishes executing. An expression e is atomic if it steps to a value in a single execution step.

By putting the $l \mapsto -$ assertion inside of an invariant I, we can thus allow both threads to access l by using HT-INV-OPEN during the faa step. A standard technique in the CSL literature is to use *ghost state* to encode a kind of *protocol* in the invariant assertion that tracks how l can evolve through the shared access by the two threads [27]. We omit the exact details of how this ghost state encoding works, but at a high level, this invariant assertion would have a format like:

$$I \triangleq \underbrace{(l \mapsto 0 * \ldots)}_{\text{no thread added}} \vee \underbrace{(\exists v.\, l \mapsto v * \ldots)}_{\text{1 thread added}} \vee \underbrace{(\exists v.\, l \mapsto v * \ldots)}_{\text{2 threads added}}$$

where threads use ghost state to track which case of this disjunction they are in.

Our key observation is that if we now also include error credits in the invariant, then we can additionally resolve the first issue mentioned above related to not having a sufficient number of error credits. For example, by setting the invariant to a form like

$$\begin{aligned} I \triangleq \Big(& \underbrace{(l \mapsto 0 * \ldots)}_{\text{no thread added}} \vee \underbrace{(\exists v.\, l \mapsto v * \ldots)}_{\text{1 thread added}} \vee \underbrace{(\exists v.\, l \mapsto v * v > 0 * \ldots)}_{\text{2 threads added}} \Big) \\ * \Big(& \underbrace{(\lightning(1/16) * \ldots)}_{\text{no thread sampled}} \vee \underbrace{(\lightning(1/4) * \ldots)}_{\text{1 thread sampled 0}} \vee \underbrace{(\lightning(1) * \ldots)}_{\text{2 threads sampled 0}} \vee \underbrace{(\lightning(0) * \ldots)}_{\geq \text{1 thread sampled non-0}} \Big) \end{aligned}$$

We initially have that the invariant owns the whole error credit $\lightning(1/16)$. The first thread to sample will use this $\lightning(1/16)$ with HT-RAND-EXP, ending up with $\lightning(1/4)$ in the case it samples 0 and $\lightning(0)$ otherwise. If the first thread samples a non-zero value, then the final value in l will be at least 0, no matter what the second thread samples. On the other hand, if the first thread samples 0, then it will return $\lightning(1/4)$ to the invariant, which will then be used by the second thread with HT-RAND-EXP to get $\lightning(1)$ in the case that it also samples 0. At that point, we can use ERR-1 to exclude this case, just as we did in the original sequential example. Of course, additional ghost state is needed to track this more complex protocol, but modern CSLs like Iris already provide sophisticated tools for encoding these kinds of protocols.

Although this example seems simple, to the best of our knowledge, Coneris is the *first* unary program logic for randomized concurrent programs that can prove this bound of $1/16$ for *conTwoAdd*. Prior concurrent separation logics, even those that are specific to first-order languages [17, 38] lack logical facilities necessary for expressing this non-trivial protocol on the shared state.

Modularity and Randomized Logical Atomicity. While placing error credits in a shared invariant solved the problems discussed in the previous part, the solution we have shown so

$$createCntr \triangleq \lambda\_.\ \mathsf{ref}\ 0 \qquad readCntr \triangleq \lambda\, l.\ !l \qquad incrCntr \triangleq \lambda\, l.\ \mathsf{faa}\, l\ (\mathsf{rand}\ 3)$$

$$conTwoAdd \triangleq \mathsf{let}\ l = createCntr\ ()\ \mathsf{in}\ (incrCntr\ l \mid\mid\mid incrCntr\ l)\,;\ readCntr\ l$$

Fig. 2. Refactored *conTwoAdd* code.

far is not modular. To see this issue, imagine that we refactored the code of *conTwoAdd* as in Figure 2. Here we introduce intermediate functions that encapsulate the operations performed on the location l. Ideally, we ought to be able to derive separate specifications for these operations, and then use only their specifications to prove the same Hoare triple we had before for *conTwoAdd*. For example, we might introduce a predicate of the form $counter\ l\ n \triangleq l \mapsto n$ capturing the value of the counter. Then, a specification for *incrCntr* might look like

$$\frac{\mathbb{E}_{\mathfrak{U}3}[\mathcal{F}] \le \varepsilon}{\{counter\ l\ x * \lightning(\varepsilon)\}\ incrCntr\ l\ \{\exists n.\ counter\ l\ (x+n) * \lightning(\mathcal{F}(n)) * n \in \{0..3\}\}}$$

which expresses the effects of adding values to the counter and also allows for averaging error credits across the different outcomes, similarly to HT-RAND-EXP. However, this specification for *incrCntr* is not sufficient for verifying *conTwoAdd*. As we saw previously for that example, we must put the points-to assertion for l (corresponding to *counter*) and the $\lightning(1/16)$ in an invariant. But we cannot use HT-INV-OPEN to open this invariant when using the above specification for *incrCntr*, because *incrCntr l* is not an atomic expression.

For *non*-probabilistic concurrent programs, a standard solution to this problem is to derive a specification that captures that *incrCntr* behaves *as if* it were atomic when incrementing the value in the counter. Although several different techniques have been proposed for encoding what it means for a function to be *logically* atomic, in Coneris we adapt the HOCAP [34] approach. At its core, the idea behind HOCAP is to observe that what makes a physically atomic expression special, in terms of the rules of the logic, is that we can open an invariant around it. Thus, to capture that an operation behaves as if it is atomic, we need a specification style that allows for opening an invariant at the logical point where an operation takes effect. In Iris, this is captured through the *update modality*, written $\Rrightarrow Q$ which holds when Q can be derived by opening invariants. By deriving a specification for *incrCntr* in which the *counter* assertion occurs under such an update modality in the pre-condition, we can enable *incrCntr* to open invariants to get this *counter* assertion at the moment when it performs the faa.

To extend this idea to the probabilistic setting, Coneris introduces a new modality, called the *probabilistic update modality* $\leadsto$, which additionally allows for error credits to be updated in an expectation preserving way, in the style of HT-RAND-EXP. Intuitively, $\leadsto P$ holds if we can make an instantaneous probabilistic update of our resources such that the outcome satisfies P. Compared to the standard update modality $\Rrightarrow$, the probabilistic update modality can additionally *redistribute* errors credits through a logical operation called *tape presampling* that allows clients to reason about future probabilistic choices. By using this new modality in HOCAP-style specifications, we are able to capture *randomized* logical atomicity, enabling modular reasoning about concurrent probabilistic libraries. Because this technique requires more advanced rules of Coneris, we postpone demonstrating it to §5.1, and first present more of the formal details of Coneris and the semantics of ConcRandML, the concurrent programming language used to express our examples.

3 Preliminaries

In §3.1, we first recall various definitions from probability theory. We then present the syntax of ConcRandML in §3.2 and its operational semantics in §3.3.

3.1 Probability Theory

To account for possibly non-terminating behavior of programs, we define our operational semantics using probability *sub*-distributions. A *discrete subdistribution* (henceforth simply *distribution*) on a countable set A is a function $\mu : A \to [0,1]$ such that $\sum_{a\in A} \mu(a) \leq 1$. The collection of distributions on A is denoted by $\mathcal{D}(A)$. The *null distribution* $\mathbf{0} : \mathcal{D}(A)$ is the constant function $\lambda x.0$. We let $\{N..M\}$ denote the set $\{n \in \mathbb{N} \mid N \leq n \leq M\}$ and for $N \geq 0$ we let $\mathfrak{U}N \colon \mathcal{D}(\mathbb{N})$ denote the (uniform) distribution that returns $1/(N+1)$ for every $n \in \{0..N\}$ and 0 otherwise. The *expected value* of $X \colon A \to [0,1]$ with respect to μ is defined as $\mathbb{E}_\mu[X] \triangleq \sum_{a\in A} \mu(a) \cdot X(a)$. The *mass* of μ is $\mathbb{E}_\mu[\lambda x.1]$. Given a predicate P on A, the *Iverson bracket* $[P]$ evaluates to 1 if $P(a)$ is true and to 0 otherwise, and the probability of P w.r.t. μ is $\Pr_\mu[P] \triangleq \mathbb{E}_\mu[[P]]$. Distributions form a monad; we write $\mu \gg\!= f$ for $\mathrm{bind}(f,\mu)$, which is defined as follows.

$$\mathrm{ret} \colon A \to \mathcal{D}(A) \qquad\qquad \mathrm{bind} \colon (A \to \mathcal{D}(B)) \times \mathcal{D}(A) \to \mathcal{D}(B)$$
$$\mathrm{ret}(a)(a') \triangleq [a = a'] \qquad\qquad \mathrm{bind}(f,\mu)(b) \triangleq \sum_{a\in A} \mu(a) \cdot f(a)(b)$$

3.2 The ConcRandML Language

Our examples are written in the ConcRandML language, which is an ML-style programming language extended with probabilistic sampling and fork-based concurrency[2]. The syntax of the language is defined by the grammar below:

$$\begin{aligned} v, w \in \mathit{Val} &::= z \in \mathbb{Z} \mid b \in \mathbb{B} \mid () \mid \ell \in \mathit{Loc} \mid \mathsf{rec}\ \mathsf{f}\ \mathsf{x} = e \mid (v, w) \mid \mathsf{inl}\, v \mid \mathsf{inr}\, v \mid \\ e \in \mathit{Expr} &::= v \mid \mathsf{x} \mid e_1\ e_2 \mid e_1 + e_2 \mid e_1 - e_2 \mid \ldots \mid \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid (e_1, e_2) \mid \mathsf{fst}\, e \mid \ldots \\ &\quad \mathsf{ref}\ e \mid\ !\, e \mid e_1 \leftarrow e_2 \mid e_1[e_2] \mid \mathsf{rand}\, e \mid \mathsf{fork}\, e \mid \mathsf{faa}\, e_1\ e_2 \\ \sigma \in \mathit{State} &::= \mathit{Loc} \xrightarrow{\text{fin}} \mathit{Val} \\ \rho \in \mathit{Cfg} &::= \mathit{List}(\mathit{Expr}) \times \mathit{State} \end{aligned}$$

The syntax is mostly standard, for example, the expressions $\mathsf{ref}\, e$, $!\, e$, and $e_1 \leftarrow e_2$ allocate, load from, and store into a reference, respectively. An array e_1 can be accessed at offset e_2 (for load or store) via $e_1[e_2]$ and $\mathsf{rand}\, N$ samples from the uniform distribution on $\{0, \ldots, N\}$.

Concurrency is supported via $\mathsf{fork}\, e$, which executes e in a new thread, and *atomic* operations which provide synchronization between threads. For example, the atomic fetch-and-add $\mathsf{faa}\, e_1\ e_2$ instruction adds the integer e_2 to the value v stored at location e_1 and returns v.[3]

A program configuration $\rho \in \mathit{List}(\mathit{Expr}) \times \mathit{State}$ is given by a pair containing the list of currently executing threads and the heap modeled as a finite map from locations to values. A configuration ρ is *final* if the first expression in the thread list is a value.

3.3 Operational Semantics

The operational semantics of ConcRandML programs is given in stages: expressions take a single execution step, which gets lifted to thread pools by schedulers, and finally these steps are chained together to obtain full program execution.

Expressions. The step function takes an expression (representing the currently active thread) and the current state and produces a distribution over the new expression, new state, and a (possibly empty) list of newly spawned threads. ConcRandML has a standard call-by-value semantics where

[2] ConcRandML is also the language studied in Polaris [35], but we consider probabilistic schedulers in addition to deterministic ones for the full program execution. We discuss this more in §8.

[3] ConcRandML also supports other atomic instructions such as atomic exchange and compare-and-swap, which we omit here for brevity.

steps can occur under evaluation contexts. Deterministic language constructs like if-then-else (1) or fork e (2) step deterministically by using the return of the distribution monad. The rand N instruction (3) uniformly associates probability $1/(N+1)$ to any integer n between 0 and N.

$$\mathsf{step} : (\mathit{Expr}, \mathit{State}) \to \mathcal{D}(\mathit{Expr}, \mathit{State}, \mathit{List}(\mathit{Expr}))$$

$$\mathsf{step}(\text{if true then } e_1 \text{ else } e_2, \sigma) = \mathrm{ret}(e_1, \sigma, []) \tag{1}$$

$$\mathsf{step}(\text{fork } e, \sigma) = \mathrm{ret}((), \sigma, [e]) \tag{2}$$

$$\mathsf{step}(\text{rand } N, \sigma) = \lambda\,(n, \sigma, []).\ \tfrac{1}{N+1} \text{ if } n \in \{0, \ldots, N\} \quad \text{and } 0 \text{ otherwise} \tag{3}$$

Thread Pools and Schedulers. The operational semantics of a configuration $\rho = (\vec{e}, \sigma)$ is then given simply by indicating which thread amongst $\vec{e}$ should step, *i.e.*, by specifying an index $i \in [0, |\vec{e}| - 1]$ and applying the step function to (e_i, σ):

$$\mathsf{tpStep}(\vec{e}, \sigma)(i) \triangleq \begin{cases} \mathbf{0} & \text{if } (\vec{e}, \sigma) \text{ is final,} \\ \mathrm{ret}(\vec{e}, \sigma) & \text{if } e_i \text{ is a value,} \\ \mathsf{step}(e_i, \sigma) \gg\!= \lambda\,(e_i', \sigma', \vec{e}').\ \mathrm{ret}(\vec{e}[i \mapsto e_i'] +\!\!+ \vec{e}', \sigma') & \text{otherwise.} \end{cases}$$

If ρ is final, it does not step. If e_i is a value, we take a stutter step. Otherwise, we update the i-th thread with the stepped expression e_i' and append the newly spawned threads $\vec{e}'$ to the thread pool.

A *scheduler* decides which thread in a configuration to step next. Formally, a (probabilistic, stateful) scheduler is given by a transition function $\zeta : (\mathit{SchedSt} \times \mathit{Cfg}) \to \mathcal{D}(\mathit{SchedSt} \times \mathbb{N})$, which takes in an internal state $\Xi \in \mathit{SchedSt}$ and a configuration ρ, and returns a distribution on its new internal state and the index of the thread in ρ to step next.

Given a configuration ρ, a scheduler ζ, and a scheduler state Ξ, we can now define the single scheduler-step reduction function $\mathsf{schStep}_\zeta(\Xi, \rho) \in \mathcal{D}(\mathit{SchedSt} \times \mathit{Cfg})$ as follows:

$$\mathsf{schStep}_\zeta(\Xi, \rho) \triangleq \zeta(\Xi, \rho) \gg\!= \lambda\,(\Xi', i).\ \mathsf{tpStep}(\rho, i) \gg\!= \lambda\,\rho'.\ \mathrm{ret}(\Xi', \rho')$$

Intuitively, $\mathsf{schStep}_\zeta(\Xi, \rho)$ first evaluates $\zeta(\Xi, \rho)$ to get a new state Ξ' and index i, steps the i-th thread to obtain the configuration ρ', and returns the new scheduler state and configuration.

The notion of scheduler we consider is quite strong. Firstly, the scheduler is *probabilistic* and can update its internal state and choose the next thread to step probabilistically instead of deterministically. Secondly, the update decision of a scheduler can depend not only on its internal state, but also on the *entire* view of the thread pool and the memory state. These two design choices provide more power to the scheduler and enable us to reason about the error bounds of algorithms under a larger and richer class of schedulers than, say, deterministic schedulers.

Program Execution. We next define n-step program execution with respect to a scheduler ζ as the following recursive function $\mathsf{exec}_{\zeta,n} : (\mathit{SchedSt} \times \mathit{Cfg}) \to \mathcal{D}(\mathit{Val})$.

$$\mathsf{exec}_{\zeta,n}(\Xi, \rho) \triangleq \begin{cases} \mathrm{ret}\, v & \text{if } \rho \text{ is final and } \rho = (v \cdot \vec{e}, \sigma) \text{ for some } v \in \mathit{Val}, \\ \mathbf{0} & \text{if } n = 0 \text{ and } \rho \text{ is not final,} \\ \mathsf{schStep}_\zeta(\Xi, \rho) \gg\!= \mathsf{exec}_{\zeta,n-1} & \text{otherwise.} \end{cases}$$

One can read $\mathsf{exec}_{\zeta,n}(\Xi, \rho)(v)$ as the probability of returning v in the first thread after at most n steps of ρ under the scheduler ζ initialized with the scheduler state Ξ. Finally, full program execution is defined as the limit of $\mathsf{exec}_{\zeta,n}$, which exists by monotonicity and continuity:

$$\mathsf{exec}_\zeta(\Xi, \rho) \triangleq \lim_{n \to \infty} \mathsf{exec}_{\zeta,n}(\Xi, \rho)$$

We simply write $\mathsf{exec}_\zeta\, e$ if the result is the same for all initial program and scheduler states.

Traditionally, a program ρ is *safe* if it never gets stuck during execution, *i.e.*, any partial program execution starting from ρ is either a value or it can make progress. To define the appropriate notion of safety for the probabilistic setting of ConcRandML (see Theorem 4.2), we need the following auxiliary definition of partial program execution $\mathrm{pexec}_{\zeta,n} : (\mathit{SchedSt} \times \mathit{Cfg}) \to \mathcal{D}(\mathit{SchedSt} \times \mathit{Cfg})$.

$$\mathrm{pexec}_{\zeta,n}(\Xi, \rho) = \begin{cases} \mathrm{ret}(\Xi, \rho) & \text{if } \rho \text{ is final or } n = 0, \\ \mathrm{schStep}_{\zeta}(\Xi, \rho) \gg\!= \mathrm{pexec}_{\zeta,n-1} & \text{otherwise.} \end{cases}$$

We can view pexec as a relaxation of exec which keeps probability mass on configurations that are not final, whereas the latter only considers final configurations.

4 Logic

In this section, we dive into the rules of Coneris. We start with a glance of the syntax of the logic and its adequacy theorem. Then, we explore the general program logic rules before discussing presampling tapes and the probabilistic update modality.

4.1 Introduction to Coneris

The Coneris logic is built on top of the Iris base logic [25] and inherits all of the basic propositions and their associated proof rules. This includes the *later* modality $\triangleright$, the *persistence* modality $\square$ and the points-to connective $\ell \mapsto v$ that asserts exclusive ownership of the location ℓ storing value v. A selection of Coneris propositions are shown below.

$$P, Q \in \mathit{iProp} ::= \mathsf{True} \mid \mathsf{False} \mid P \wedge Q \mid P \vee Q \mid P \Rightarrow Q \mid \forall x.\, P \mid \exists x.\, P \mid P * Q \mid P \mathrel{-\!\!*} Q \mid \triangleright P \mid \square P \mid$$
$$\boxed{P}^{\iota} \mid \boxed{g}^{\gamma} \mid {}_{\mathcal{E}_1}\!\Vdash_{\mathcal{E}_2} P \mid \ell \mapsto v \mid \{P\}\, e\, \{Q\}_{\mathcal{E}} \mid \lightning(\varepsilon) \mid \kappa \hookrightarrow (N, \vec{n}) \mid {}_{\mathcal{E}_1}\!\rightsquigarrow_{\mathcal{E}_2} P \mid \ldots$$

Coneris is a separation logic and propositions denote sets of resources. $P * Q$ holds for resources that can be decomposed into two disjoint pieces satisfying P and Q. The separating implication $P \mathrel{-\!\!*} Q$ is the right adjoint of $*$, in the sense that $P * (P \mathrel{-\!\!*} Q) \vdash Q$. While omitted in §2, note that invariant assertions $\boxed{P}^{\iota}$ are annotated with an identifying name ι which is used to prevent the prover from opening the same invariant twice (which is unsound). For bookkeeping purposes, Hoare triples are annotated with the set of invariant names that the specification relies on; we omit this *mask* annotation when considering the set of all invariant names $\top$.

As mentioned in §2, we internalize error bounds using the error credit assertion $\lightning(\varepsilon)$. The *presampling tape* assertion $\kappa \hookrightarrow (N, \vec{n})$ is a probabilistic connective that we adapt from Clutch [20] which plays a key role in deriving certain modular specifications. The probabilistic update modality ${}_{\mathcal{E}_1}\!\rightsquigarrow_{\mathcal{E}_2} P$ is a novelty of the Coneris logic. We further discuss these three connectives and their role in the following section.

The meaning of the Coneris Hoare triple is captured by the adequacy theorem shown below.

Theorem 4.1 (Adequacy). *If* $\{\lightning(\varepsilon)\}\, e\, \{\phi\}$, *then for all schedulers* ζ, $\Pr_{\mathrm{exec}_{\zeta} e}[\neg\phi] \leq \varepsilon$.

The theorem says that by proving a Hoare triple for the expression e, assuming initial ownership of $\lightning(\varepsilon)$ error credits, then for all schedulers ζ, the probability of the program e returning a value *not* satisfying the proposition ϕ is smaller than or equal to ε.

In addition, we have another safety theorem that provides an upper bound on the probability of the expression getting stuck.

Theorem 4.2 (Safety). *If* $\{\lightning(\varepsilon)\}\, e\, \{\mathsf{True}\}$, *then for all schedulers* ζ *with mass* 1 *and integers* n, *the mass of* $\mathrm{pexec}_{\zeta,n}(\Xi, ([e], \sigma))$ *is greater or equal to* $1 - \varepsilon$.

Intuitively, this theorem states that proving $\{\lightning(\varepsilon)\}\ e\ \{\text{True}\}$ in Coneris implies that the probability of e getting stuck is at most ε for all schedulers[4].

4.2 Rules of Coneris

Program-Logic Rules. Coneris satisfies all the usual structural and computational rules present in Iris-based separation logics. For example, Coneris satisfies the bind rule (HT-BIND), the frame rule (HT-FRAME), and the usual computational rules of for interacting with the heap (*e.g.*, HT-LOAD).

HT-BIND
$$\frac{\{P\}\ e\ \{v.Q\} \qquad \forall v.\ \{Q\}\ K[v]\ \{R\}}{\{P\}\ K[e]\ \{R\}}$$

HT-FRAME
$$\frac{\{P\}\ e\ \{Q\}}{\{P * R\}\ e\ \{Q * R\}}$$

HT-LOAD
$$\frac{}{\{l \mapsto v\}\ !l\ \{w.w = v * l \mapsto v\}}$$

Invariants can be allocated by giving up ownership of the corresponding resources (HT-INV-ALLOC). If we own an invariant, we can temporarily, for one atomic step, get access to its contents (HT-INV-OPEN). The later modality $\triangleright$ is important for soundness but can otherwise be ignored [25].

HT-INV-ALLOC
$$\frac{\{\boxed{P}^{\iota} * Q\}\ e\ \{R\}_{\mathcal{E}}}{\{P * Q\}\ e\ \{R\}_{\mathcal{E}}}$$

HT-INV-OPEN
$$\frac{e\ \text{atomic} \qquad \{\triangleright I * P\}\ e\ \{\triangleright I * Q\}_{\mathcal{E}}}{\{\boxed{I}^{\iota} * P\}\ e\ \{Q\}_{\mathcal{E}\uplus\{\iota\}}}$$

The Update Modality. The *update modality* ${}_{\mathcal{E}_1}\!\Rightarrow_{\mathcal{E}_2}$ is the primary primitive for manipulating ghost resources and interacting with invariants in the Iris base logic. As we alluded to earlier in §2, a key idea behind the HOCAP approach to modular specification is to use this modality as a way to assert that a proposition could be proven by opening invariants.

The update modality ${}_{\mathcal{E}_1}\!\Rightarrow_{\mathcal{E}_2}$ is annotated with two sets of invariants. We write $\Rightarrow_{\mathcal{E}}$ when $\mathcal{E}_1 = \mathcal{E}_2 = \mathcal{E}$ and $\Rightarrow$ when $\mathcal{E} = \top$, the set of all names. Intuitively, the assertion ${}_{\mathcal{E}_1}\!\Rightarrow_{\mathcal{E}_2} P$ denotes a resource that, together with the resources from the invariants in $\mathcal{E}_1$, can be updated and split into two disjoint pieces: one satisfying P and one satisfying the invariants in $\mathcal{E}_2$. That is, we can use the update modality to *specify* resource updates and invariant access (INV-OPEN) as an *assertion* in the logic rather than just as a primitive rule of the program logic. The update modality can be eliminated (HT-FUPD-ELIM) at any suitable time during program verification.

INV-OPEN
$$\frac{\boxed{P}^{\iota} \qquad \triangleright P \mathrel{-\!\!*} {}_{\mathcal{E}_1}\!\Rightarrow_{\mathcal{E}_2} (\triangleright P * Q)}{{}_{\mathcal{E}_1\uplus\{\iota\}}\!\Rightarrow_{\mathcal{E}_2\uplus\{\iota\}} P}$$

HT-FUPD-ELIM
$$\frac{e\ \text{atomic} \qquad \{P * Q\}\ e\ \{{}_{\mathcal{E}_2}\!\Rightarrow_{\mathcal{E}_1} R\}_{\mathcal{E}_2}}{\{({}_{\mathcal{E}_1}\!\Rightarrow_{\mathcal{E}_2} P) * Q\}\ e\ \{R\}_{\mathcal{E}_1}}$$

A key idea behind the approach we apply in §5 is to parameterize program specifications by a proposition of the shape $P \mathrel{-\!\!*} {}_{\mathcal{E}_1}\!\Rightarrow_{\mathcal{E}_2} Q$, a so-called *view shift*, that is eliminated at the linearization point of the module operation. By providing a view shift as an argument, the *client* can specify how they wish for their logical state (their "view") to evolve when the operation physically takes place.

Presampling Tapes. Reminiscent of how prophecy variables [1, 2, 26] allow us to talk about the future, presampling tapes give us the means to talk about the outcome of sampling statement in the future. Presampling tapes were introduced in Clutch [20] to address an alignment issue in refinement proofs, but as we later see in §5 they also play a crucial role in modularizing (unary) proofs about concurrent probabilistic programs through probabilistic view shifts.

Intuitively, presampling tapes allow us *in the logic* to presample the outcome of future sampling statements. Formally, they appear both operationally and in the logic. In the programming language, presampling tapes appear as two new ghost code constructs, tape e and rand $e_1\ e_2$, that are used to allocate a new presampling tape and sample from a tape, respectively.

[4]The condition that ζ must have mass 1 (for all scheduler states) rules out the pathological situation where a configuration is "stuck" because ζ has probability less than 1 to pick *any* thread to step next. The assumption is only used in Theorem 4.2.

$$v \in \mathit{Val} ::= \ldots \mid \kappa \in \mathit{Label} \qquad \sigma \in \mathit{State} \triangleq (\mathit{Loc} \xrightarrow{\text{fin}} \mathit{Val}) \times (\mathit{Label} \xrightarrow{\text{fin}} \mathit{Tape})$$
$$e \in \mathit{Expr} ::= \ldots \mid \mathsf{tape}\ e \mid \mathsf{rand}\ e_1\ e_2 \qquad t \in \mathit{Tape} \triangleq \{(N, \vec{n}) \mid N \in \mathbb{N} \land \vec{n} \in \mathbb{N}^*_{\leq N}\}$$

In the operational semantics, allocation of a fresh presampling tape (4) via $\mathsf{tape}\, N$ deterministically associates a fresh label κ to the empty tape ϵ. A labelled $\mathsf{rand}\, \kappa\ N$ with an empty tape samples uniformly (5), *i.e.*, it behaves like an unlabelled $\mathsf{rand}\, N$. If, on the other hand, the tape κ is non-empty, $\mathsf{rand}\, N\ \kappa$ deterministically pops the first value n from the tape (6). Note that no step in the operational semantics *writes* contents to a presampling tape. In fact, tapes and label annotations do not in any way alter the behavior of the program and can be entirely erased [20]. However, as we later see, the probabilistic update modality allows us to reason *as if* a presampling step could asynchronously pre-populate a tape with a random sample at any point in time.

$$\mathrm{step}(\mathsf{tape}\, N, \sigma) = \mathrm{ret}(\kappa, \sigma[\kappa := (N, \epsilon)], []) \quad (\text{where } \kappa \text{ is fresh w.r.t. } \sigma) \tag{4}$$
$$\mathrm{step}(\mathsf{rand}\, \kappa\ N, \sigma) = \lambda\, (n, \sigma, []).\ \tfrac{1}{N+1} \text{ if } \sigma[\kappa] = (N, \epsilon) \land n \in \{0, \ldots, N\} \quad \text{and} \quad 0 \text{ otherwise} \tag{5}$$
$$\mathrm{step}(\mathsf{rand}\, \kappa\ N, \sigma[\kappa := n \cdot \vec{n}]) = \mathrm{ret}(n, \sigma[\kappa := \vec{n}], []) \tag{6}$$

Now, the logical assertion $\kappa \hookrightarrow (N, \vec{n})$ denotes ownership of the presampling tape κ with bound N and contents $\vec{n}$, analogously to how the points-to connective for the heap denotes ownership of a location and its contents. The two rules HT-ALLOC-TAPE and HT-RAND-TAPE reflects the operational behavior of equations (4) and (6) in the logic.

HT-ALLOC-TAPE
$$\frac{}{\{\mathsf{True}\}\ \mathsf{tape}\ N\ \{\kappa.\, \kappa \hookrightarrow (N, \epsilon)\}}$$

HT-RAND-TAPE
$$\frac{}{\{\kappa \hookrightarrow (N, n \cdot \vec{n})\}\ \mathsf{rand}\, \kappa\ N\ \{x.\, x = n * \kappa \hookrightarrow (N, \vec{n})\}}$$

Probabilistic Update Modality. Previous work [3, 20, 21] introduce presampling tapes for different purposes but, common for all instantiations, presampling is only supported as a rule in the program logic. Similar to how HT-INV-OPEN only allows us to reason about invariants using the program logic, presampling is only supported as a primitive program-logic rule. This is *not* sufficient for the modular specifications we set out to prove. Intuitively, we need a way to specify updates to presampling tapes as an *assertion*, just like the update modality enables us to specify invariant access and resource updates as an assertion.

To this end, we introduce the *probabilistic update modality* ${}_{\mathcal{E}_1}\!\rightsquigarrow_{\mathcal{E}_2} P$. This modality satisfies all the same rules as the update modality: *e.g.*, it can be used to open invariants (hence the invariant masks $\mathcal{E}_1$ and $\mathcal{E}_2$) and update resources, it is monadic (PUPD-RET and PUPD-BIND), can be derived from the update modality (PUPD-FUPD), and it can be eliminated (HT-PUPD-ELIM).

PUPD-RET
$$\frac{P}{\rightsquigarrow_{\mathcal{E}} P}$$

PUPD-BIND
$$\frac{{}_{\mathcal{E}_1}\!\rightsquigarrow_{\mathcal{E}_2} P \qquad P \mathrel{-\!\!*} {}_{\mathcal{E}_2}\!\rightsquigarrow_{\mathcal{E}_3} Q}{{}_{\mathcal{E}_1}\!\rightsquigarrow_{\mathcal{E}_3} Q}$$

PUPD-FUPD
$$\frac{{}_{\mathcal{E}_1}\!\Rightarrow_{\mathcal{E}_2} P}{{}_{\mathcal{E}_1}\!\rightsquigarrow_{\mathcal{E}_2} P}$$

HT-PUPD-ELIM
$$\frac{\{P * Q\}\ e\ \{R\}_{\mathcal{E}}}{\{(\rightsquigarrow_{\mathcal{E}} P) * Q\}\ e\ \{R\}_{\mathcal{E}}}$$

PUPD-PRESAMPLE-EXP
$$\frac{\mathbb{E}_{\mathfrak{U}N}[\mathcal{F}] \leq \varepsilon \qquad \lightning(\varepsilon) \qquad \kappa \hookrightarrow (N, \vec{n})}{\rightsquigarrow_{\mathcal{E}} (\exists n.\, \kappa \hookrightarrow (N, \vec{n} \cdot n) * \lightning(\mathcal{F}(n)) * n \in \{0..N\})}$$

PUPD-ERR
$$\frac{}{\rightsquigarrow_{\mathcal{E}} (\exists \varepsilon.\, 0 < \varepsilon * \lightning(\varepsilon))}$$

The key novelty of the probabilistic modality is its ability to populate presampling tapes as shown in PUPD-PRESAMPLE-EXP. The rule says that if we own a presample tape we can populate the tape with a freshly sampled value n. Similar to HT-RAND-EXP, it allows re-distributing error credits along different branches of the randomized outcome, as long as the expected value of the error credit does not increase. We showcase the probabilistic update modality on an example in §5.4.

As a somewhat orthogonal property, the probabilistic update modality also internalizes the notion of continuity of probabilities within the logic. Specifically, it permits synthesizing some arbitrarily small error credit *out of thin air* as seen in PUPD-ERR. This principle enables *induction by error amplification* [3] as we showcase in §5.4.

5 Modular Specifications of Concurrent Randomized Modules

In this section, we first provide an overview of how HOCAP-style specifications capture logically atomicity of concurrent data structures. Next, we explain how we extend the approach to capture *randomized* logical atomicity and present a modular specification for the randomized counter module. We also describe how the specification is strong enough to verify clients that use the randomized counter module concurrently. Subsequently, we present three different implementations of the concurrent randomized counter module and discuss how to show that they satisfy the specification. Later in §6 we discuss how we verify a series of larger case studies.

5.1 Modular Specifications of Concurrent Randomized Modules: Overview

Before considering the randomized setting, we showcase our specification style on a non-randomized example: a concurrent counter module with functions for creating a counter, (deterministically) incrementing by one, and reading.

As alluded to in §4.2, the high level idea is to parameterize specifications by a view shift that captures how the logical state of the counter evolves at the linearization point. Our specification of the non-randomized counter module is shown in Figure 3.

$$\{\mathsf{True}\}\ \mathit{createCntr}()\ \{c.\ \exists \iota.\ \mathit{counter}\ \iota\ c * \mathit{cfrag}\ 1\ 0\}$$

$$\forall \mathcal{E}, \iota, c, Q.\ \left\{\mathit{counter}\ \iota\ c * (\forall z.\ \mathit{cauth}\ z \mathrel{-\!\!*} \Rrightarrow_{\mathcal{E}} \mathit{cauth}\ (z+1) * Q\ z)\right\}\ \mathit{incrCntr}\ c\ \{z.\ Q\ z\}_{\mathcal{E} \uplus \{\iota\}}$$

$$\forall \mathcal{E}, \iota, c, Q.\ \left\{\mathit{counter}\ \iota\ c * (\forall z.\ \mathit{cauth}\ z \mathrel{-\!\!*} \Rrightarrow_{\mathcal{E}} \mathit{cauth}\ z * Q\ z)\right\}\ \mathit{readCntr}\ c\ \{z.\ Q\ z\}_{\mathcal{E} \uplus \{\iota\}}$$

Fig. 3. Specification for a (non-randomized) concurrent counter module.

When creating a counter, one obtains ownership of two resources: $\mathit{counter}\ \iota\ c$ and $\mathit{cfrag}\ 1\ 0$. The $\mathit{counter}\ \iota\ c$ resource captures that c is a counter with an associated invariant name ι. Intuitively, this invariant contains the internal state of the counter but the details are unknown to clients. The predicate is persistent, *i.e.*, $\mathit{counter}\ \iota\ c \dashv\vdash \mathit{counter}\ \iota\ c * \mathit{counter}\ \iota\ c$ and can hence be freely shared.

The predicates *cauth* and *cfrag* provide *authoritative* and *fragmental* views of the counter. Intuitively, *cauth* provides the counter module's view of the counter and *cfrag* denotes the client's view. A fragmental view $\mathit{cfrag}\ q\ n$ denotes a q-fractional view that the counter is *at least* the value n. The $\mathit{cfrag}\ q\ n$ resource can be split and combined, *i.e.*, $\mathit{cfrag}\ (q_1 + q_2)\ (n_1 + n_2) \dashv\vdash \mathit{cfrag}\ (q_1, n_1) * \mathit{cfrag}(q_2, n_2)$ and thus shared. The fragmental view is guaranteed to be consistent with the authoritative view, *i.e.*, $\mathit{cauth}\ n * \mathit{cfrag}\ q\ m \vdash m \leq n$ and $\mathit{cauth}\ n * \mathit{cfrag}\ 1\ m \vdash m = n$, and updated accordingly, *i.e.*, $\mathit{cauth}\ n * \mathit{cfrag}\ q\ m \vdash \Rrightarrow_{\mathcal{E}} \mathit{cauth}\ (n+p) * \mathit{cfrag}\ q\ (m+p)$.

The specification for the increment and read functions are parameterized by a view shift that gives (temporary) access to the module's view. This is one of the key ideas of HOCAP-style specifications. From the client's perspective, the view shift is a proof obligation. For the increment function, proving this view shift requires having ownership of a fragmental view (to update the resources), but the fragmental view can be provided by opening an invariant using the update modality. The client-chosen predicate Q lets the client derive information as part of the view shift. For example, they can pick $Q\ z \triangleq \mathit{cfrag}\ q\ (n+1) \wedge z = (n+1)$.

Probabilistic Concurrent modules with Error Redistribution. Now, consider the randomized concurrent counter module from §2 where the increment function increments the counter by a value chosen uniformly at random from 0 to 3. For the client to be able to redistribute error credits as part of the random sampling, we parameterize the specification of the increment function by another view shift as shown in Figure 4.

$$\forall \mathcal{E}, \iota, c, Q.\ \left\{ \textit{counter}\ \iota\ c * {}_{\mathcal{E}}\Rrightarrow_{\emptyset} \left(\begin{array}{r} \exists \varepsilon, \mathcal{F}.\ \lightning(\varepsilon) * (\mathbb{E}_{\mathfrak{U}3}[\mathcal{F}] \le \varepsilon) * \forall x \in \{0..3\}.\ \lightning(\mathcal{F}(x)) \mathrel{-\!\!*} \\ \left({}_{\emptyset}\Rrightarrow_{\mathcal{E}} (\forall z.\ \textit{cauth}\ z \mathrel{-\!\!*} \Rrightarrow_{\mathcal{E}} \textit{cauth}\ (z + x) * Q\ \varepsilon\ \mathcal{F}\ x\ z)\right) \end{array} \right) \right\}$$
$$\textit{incrCntr}\ c$$
$$\{z.\ \exists \varepsilon, \mathcal{F}, x.\ Q\ \varepsilon\ \mathcal{F}\ x\ z\}_{\mathcal{E} \uplus \{\iota\}}$$

Fig. 4. Specification of *incrCntr* for a concurrent randomized counter module.

Notice that in the precondition the client now has to prove a view shift which is split into two parts. We begin by looking at the second part (the line at the bottom). This is analogous to the deterministic case, except that the abstract state *cauth z* gets incremented by some uniformly sampled $x \in \{0..3\}$. This operation is randomized, so we also let the client update their error credits along this distribution, which is the first part of the view shift. After opening all invariants in $\mathcal{E}$, the client chooses some ε and an error distribution function $\mathcal{F}$, gives up $\lightning(\varepsilon)$, gets back $\lightning(\mathcal{F}(x))$, re-establishes all invariants in $\mathcal{E}$, and goes on to prove the second part. Notice that the specification allows the client to retrieve error credits from an invariant. Intuitively, these two parts of the view shift capture two separately logically atomic actions of the increment operation. The first being the random operation where we re-distribute errors, and the second being the actual increment, where we increase the counter by the sampled value. If all these preconditions are satisfied, then at the end of *incrCntr*, we return some value z which satisfies $Q\ \varepsilon\ \mathcal{F}\ x\ z$ for some ε, $\mathcal{F}$, and x. The specification for creating and reading the counter are unchanged as no randomization is involved.

Probabilistic Concurrent Modules with Error Redistribution and Presampling. One limitation of the previous specification is that the sampling operation is fixed to take place within the function call *incrCntr*. As a result, the only point at which randomness can be generated for the module, and errors can be distributed, is at the invocation of the increment operation. However, it is sometimes useful to reason about the probabilistic part of the operation asynchronously.

In Clutch [20] presampling tapes are used to generate randomness asynchronously and facilitate refinement proofs. In a concurrent setting, there is also an asynchronous component arising from the order in which randomized operations are physically resolved, and we propose the use of presampling and tapes to resolve them in advance and independently from this order.

In the previous specification, the view shift consisted of a probabilistic part (*i.e.*, spending and distribution of error credits) and a deterministic part (updating the abstract state). Presampling allows us to decouple these two parts and reason about them separately, resulting in a more expressive HOCAP-style specification. We demonstrate that by exposing tapes and presampling operations in the module specifications, clients can perform presampling for an abstract randomized operation. This is an *indispensable* technique for verifying certain concurrent modules, and we show an example in §5.2.

The new and final specification of the probabilistic counter module, which includes not only error redistribution, but also a (ghost) method for creating an abstract presampling tape and a (ghost) operation for sampling on a tape, see Figure 5. This specification has a new predicate *ctape* that stores the presampled randomness for the random counter. Note that *ctape* is an abstract predicate which might be realized in multiple ways besides using primitive tape predicates, which

$$\forall \mathcal{E}, \varepsilon, \mathcal{F}, \vec{n}, \kappa.\ (\lightning(\varepsilon) * (\mathbb{E}_{\mathcal{U}3}[\mathcal{F}] \le \varepsilon) * \mathit{ctape}\ \kappa\ \vec{n} \mathrel{-\!\!*} \pmapsto_{\mathcal{E}} \exists n \in \{0..3\}.\ \lightning(\mathcal{F}(n)) * \mathit{ctape}\ \kappa\ (\vec{n} \cdot [n]))$$

$$\forall \iota, c.\ \{\mathit{counter}\ \iota\ c\}\ \mathit{createCtape}()\ \{\kappa.\ \mathit{ctape}\ \kappa\ \epsilon\}$$

$$\forall \mathcal{E}, \iota, c, n, \vec{n}, Q.\ \left\{ \begin{array}{l} \mathit{counter}\ \iota\ c * \mathit{ctape}\ \kappa\ (n \cdot \vec{n}) * \\ (\forall z.\ \mathit{cauth}\ z \mathrel{-\!\!*} \Rrightarrow_{\mathcal{E}} \mathit{cauth}\ (z+n) * Q\ z) \end{array} \right\} \mathit{incrCntr}\ c\ \kappa\ \left\{z.\mathit{ctape}\ \kappa\ \vec{n} * Q\ z\right\}_{\mathcal{E} \uplus \{\iota\}}$$

Fig. 5. Specification for a randomized counter module with presampling tapes.

allows us to hide the details of how different implementations of the counter module physically generate randomness. By exposing the abstract presampling tape explicitly, we aim to capture more of the proof principles for a concrete randomized operation. (In §5.2, we demonstrate that this specification which exposes abstract tapes is in fact *more general* than the previous one)

Compared to the previous randomized specification, reasoning about randomness of the increment operation is now extracted into a separate condition that utilizes the probabilistic update modality ($\pmapsto P$), which says that we can presample onto the *ctape* and distribute errors in a expectation-preserving manner. With this change, clients can allocate their own local tapes via *createCtape* and reason about randomness locally. The *incrCntr* function takes a non-empty *ctape* predicate as argument, and acts in a (logically) deterministic manner, by reading and consuming the first element n of the tape, and incrementing the abstract state of the counter by n.

Now that we have shown an expressive specification (Figure 5), in the following sections, we show how this specification suffices to verify clients. We also show that three different implementations of the probabilistic random counter module all meet this specification. These three implementations exhibit different numbers of sampling operations, but yet they all meet the same abstract module specification. In other words, the randomization of the increment operation acts "logically atomic" as expressed by a single probabilistic update, even if in reality, it is not. From the perspective of a client, random sampling within the increment operation appears to behave as if it is simply a single rand 3. We refer to this as *randomized logical atomicity*.

5.2 Verifying Clients of Randomized Counter Module

We now describe how the specification with error re-distribution and presampling tapes shown in Figure 5 can be used to verify concurrent clients. We also show how the HOCAP-style specification that exposes abstract tapes is more general than the one that does not.

Revisiting *conTwoAdd*. We begin with the *conTwoAdd* client example introduced in §2. Since the new specification of the randomized concurrent counter utilizes tapes, we annotate the *conTwoAdd* client to use the abstract tapes exposed in the specification:

$$\begin{array}{l} \mathit{conTwoAdd} \triangleq \mathsf{let}\ c = \mathit{createCntr}()\ \mathsf{in} \\ \quad \left(\begin{array}{l} \mathsf{let}\ \kappa = \mathit{createCtape}()\ \mathsf{in} \\ \mathit{incrCntr}\ c\ \kappa \end{array} \ \middle\|\ \begin{array}{l} \mathsf{let}\ \kappa = \mathit{createCtape}()\ \mathsf{in} \\ \mathit{incrCntr}\ c\ \kappa \end{array} \right); \\ \quad \mathit{readCntr}\ c \end{array}$$

Recall that we expect the return value to be 0 with a probability 1/16. We state this through the following Coneris Hoare triple: $\{\lightning(1/16)\}\ \mathit{conTwoAdd}\ \{v.v > 0\}$.

We present here a high level intuition for the proof and defer the details to Appendix A. Most of this proof is similar to the one sketched in §2 where we allocate an invariant that encodes a protocol that tracks both the available amount of error credits and the ghost state of both threads and describes how they can evolve. In the case where both threads sampled 0, we are able to obtain $\lightning(1)$ from the invariant at the end and derive a contradiction with ERR-1.

The difference between this proof and that from §2 is twofold. Firstly, the randomness is generated asynchronously using the presampling rule and the abstract tapes. The probabilistic update modality allows us to open the invariant, obtain error credits from it, presample onto our abstract tapes, redistribute the error credits, and close the invariant again, all in an atomic manner. Secondly, to apply *incrCntr* and *readCntr*, we need to prove the view shifts in the precondition of their corresponding Hoare triple specifications.

An important detail of this proof is that we do not need to place any *ctape* predicate in the invariant. Each thread uses a separate and local tape which does not need to be shared. This kind of "local tape" principle lets each thread "own" its own randomness and this simplifies the proof since we need not worry how the state of *ctape* is changed by other external concurrent threads.

Advantage of Exposing Abstract Tapes. Recall that in §5.1, we presented a simpler specification of the randomized counter module (Figure 4) that does not expose presampling tapes as abstract predicates. To see why that specification is not as general as that in Figure 5 and that it is useful to expose the presampling tapes in the specification, consider the following *twoIncr* program and its specification in Figure 6.

$$
\begin{array}{l}
\mathit{twoIncr}\ \_ \triangleq \mathsf{let}\ c = \mathit{createCntr}\ ()\ \mathsf{in} \\
\qquad \mathsf{let}\ \kappa = \mathit{createCtape}\ ()\ \mathsf{in} \\
\qquad \mathit{incrCntr}\ c\ \kappa; \\
\qquad \mathsf{let}\ v_1 = \mathit{readCntr}\ c\ \mathsf{in} \\
\qquad \mathit{incrCntr}\ c\ \kappa; \\
\qquad \mathsf{let}\ v_2 = \mathit{readCntr}\ c - v_1\ \mathsf{in} \\
\qquad 4 \cdot v_1 + v_2
\end{array}
$$

$$
\left\{ \begin{array}{l} \mathcal{E} \Rrightarrow_{\emptyset} \exists \varepsilon, \mathcal{F}.\ \lightning(\varepsilon) \ast (\mathbb{E}_{\mathfrak{U}15}[\mathcal{F}] \le \varepsilon) \ast \\ \qquad (\forall x.\ \lightning(\mathcal{F}(x)) \mathrel{-\!\!*} {}_{\emptyset}\Rrightarrow_{\mathcal{E}} Q\ \varepsilon\ \mathcal{F}\ x) \end{array} \right\}
$$
$$
\mathit{twoIncr}\ ()
$$
$$
\{z.\ \exists \varepsilon, \mathcal{F}.\ Q\ \varepsilon\ \mathcal{F}\ z\}_{\mathcal{E} \uplus \{\iota\}}
$$

Fig. 6. Implementation and specification of *twoIncr*.

The sequential program *twoIncr* first creates a new randomized counter and allocates a tape for the counter. It then performs two *incrCntr* and *readCntr* pair operations successively, to read the exact values v_1 and v_2 added to the counter. At the end it returns $4 \cdot v_1 + v_2$. As both v_1 and v_2 are sampled uniformly from $\{0, \ldots, 3\}$, the return value is uniformly distributed between $\{0, \ldots, 15\}$. This is captured by the Hoare triple in Figure 6 where error credits can be re-distributed across the 16 possibilities in an expectation-preserving way. Note that the view shift of the Hoare triple captures the fact that the re-distribution happens in a logically-atomic manner.

Proving the specification of *twoIncr* with the more general specification (Figure 5) is relatively straightforward. After applying the specification for creating the counter and the tape, we perform two consecutive presamples onto *ctape* with the presampling specification of the counter module. These two presampling operations are combined into one atomic operation with the PUPD-BIND rule, allowing us to use the view shift provided in the precondition to split the error credits for the 16 possibilities. The rest of the proof follows directly by applying the specification for incrementing the counter with the tape and reading from it twice.

However, the specification without the presampling tapes exposed (Figure 4) is not strong enough to prove this Hoare triple. The specification restricts the error redistribution to only occur within the *incrCntr* call, and we are unable to combine the two separate error redistribution operations in each *incrCntr* call into one atomic action. On the other hand, the more general specification allows us to "pull" the randomized operation out of the *incrCntr* call and perform the randomized operation in advance using the presampling operation of the abstract tapes.

5.3 Three Implementations of the Randomized Counter Module

Recall from §5.1 that the specification of the randomized counter module from Figure 5 provides four methods: *createCntr* for creating the counter, *createCtape* for creating a tape, *incrCntr* for incrementing the counter with a random value chosen uniformly from the set $\{0, \ldots, 3\}$ (sampled from the tape), and *readCntr* for reading the value of the counter.

To illustrate the expressiveness of our modular specification, we consider three implementations that we show meet the same specification, which we refer to as I_1, I_2, and I_3, respectively. They only differ in the way they implement the *createCtape* and *incrCntr* method–the implementations of *createCntr* and *readCntr* are the same in all three implementations:

$$\mathit{createCntr} \triangleq \lambda\, \_.\ \mathsf{ref}\, 0 \qquad \mathit{readCntr} \triangleq \lambda\, l.\ !\, l$$

Internally, the counter is represented by a pointer to a number and the read method simply dereferences the pointer. The three implementations of the create tape and increment methods are shown in Figure 7. In I_1, the increment method simply increments the counter value stored at the

$$\begin{aligned}
&\mathit{createCtape}_1 \triangleq \lambda\, ().\ \mathsf{tape}\, 3 && \mathit{incrCntr}_1 \triangleq \lambda\, l, \kappa.\ \mathsf{faa}\, l\, (\mathsf{rand}\, \kappa\, 3)\\
&\mathit{createCtape}_2 \triangleq \lambda\, ().\ \mathsf{tape}\, 1 && \mathit{incrCntr}_2 \triangleq \lambda\, l, \kappa.\ \mathsf{let}\, \kappa = \mathsf{tape}\, 1\, \mathsf{in}\\
&\mathit{createCtape}_3 \triangleq \lambda\, ().\ \mathsf{tape}\, 4 && \qquad\qquad \mathsf{faa}\, l\, (\mathsf{rand}\, \kappa\, 1 \cdot 2 + \mathsf{rand}\, \kappa\, 1)\\
& && \mathit{incrCntr}_3 \triangleq \mathsf{rec}\, f\, l\, \kappa = \mathsf{let}\, x = \mathsf{rand}\, \kappa\, 4\, \mathsf{in}\\
& && \qquad\qquad \mathsf{if}\, x < 4\, \mathsf{then}\ \mathsf{faa}\, l\, x\, \mathsf{else}\, f\, l\, \kappa
\end{aligned}$$

Fig. 7. Implementation of the counter module.

location by a rand 3 chosen value between 0 and 3 using a fetch-and-add instruction. The function hence creates a tape with bound 3. In I_2, the increment method is implemented using two coin flips (*i.e.*, calls to rand 1), and in I_3, we use a recursive rejection sampler that, in order to simulate rand 3, repeatedly samples from rand 4 until it gets a value within $\{0, \ldots 3\}$. The *createCtape* function for both implementations creates a tape with bound 1 and 4, respectively.

For I_2 and I_3 in particular, it is interesting that even though the implementations do not sample randomness atomically (e.g., I_3 can possibly execute any number of rand 4 operations), they still meet the specification where the presampling of a single value onto the abstract tape is described by a *single* probabilistic update modality as we show in the next section. In other words, we capture *randomized logically atomicity* of the module in the sense that externally, there appears to be a single randomized transition within the *incrCntr* function.

5.4 Verifying I_1, I_2, and I_3

We now show how the three randomized counter implementations meet the specification with error redistribution and presampling tapes. We start by giving concrete definitions for the three abstract predicates. For the three implementations, it turns out that the counter predicate *counter*, and the *cauth* and *cfrag* predicates are defined identically; the persistent counter predicate $\mathit{counter}\ \iota\ c$ is defined as $\boxed{\exists l, n.\ c = l * l \mapsto n * \mathit{cauth}\, n}^{\iota}$ and the *cauth* and *cfrag* predicates are defined with a standard authoritative-fractional resource algebra [25]. We show the exact definition of *ctape* for each of the three implementations below.

$$\begin{aligned}
&\mathit{ctape}_1\ \kappa\ \vec{n} \triangleq \kappa \hookrightarrow (3, \vec{n})\\
&\mathit{ctape}_2\ \kappa\ \vec{n} \triangleq \kappa \hookrightarrow (1, \mathsf{expand}\, \vec{n}) * (\forall x \in \vec{n}.\, x < 4)\\
&\mathit{ctape}_3\ \kappa\ \vec{n} \triangleq \exists \vec{m}.\, \mathsf{filter}\ (\lambda\, x.\, x < 4)\ \vec{m} = \vec{n} * \kappa \hookrightarrow (4, \vec{m})
\end{aligned}$$

For $ctape_1$, since the first implementation uses a rand 3 to sample from 0 to 3 directly, we define $ctape_1$ with the presampling tape $\kappa \hookrightarrow (3, \vec{n})$. For $ctape_2$, since we are sampling from 0 to 3 via two rand 1s, the predicate is defined by expanding the tape elements into its binary representation. The function expand takes in a list of numbers and rewrites them into binary representation while keeping the list "flattened". For example $\mathsf{expand}([2;3;1;0])$ returns $[1;0;1;1;0;1;0;0]$. Finally, for the third implementation, if we are logically storing $\vec{n}$ with our *ctape* predicate, the concrete tape stores some list $\vec{m}$ such that $\vec{n}$ is equal to $\vec{m}$ with all 4s removed from it.

It suffices to show that the functions *createCntr*, *createCtape*, *incrCntr*, and *readCntr* satisfy the specification and that *ctape* satisfies the presampling probabilistic update specification, *i.e.*, we can logically append a new element into the *ctape* while redistributing errors. The specification of the functions are not too complicated. As an example, consider the *incrCntr* specification for I_3.

$$\begin{array}{l}\left\{counter\ \iota\ c * ctape\ \kappa\ (n \cdot \vec{n}) * (\forall z.\ cauth\ z \mathrel{-\!\!*} \Rrightarrow_{\mathcal{E}} cauth\ (z+n) * Q\ z)\right\} \\ \quad incrCntr_3\ c\ \kappa \\ \{z.\ ctape\ \kappa\ \vec{n} * Q\ z\}_{\mathcal{E} \uplus \{\iota\}}\end{array}$$

After unfolding the definition of the abstract predicates for I_3, we repeatedly loop through the recursive function until we reach a value n in the tape that is smaller than 4 by structural induction on the tape or Löb induction. During the atomic faa operation, we open the invariant with HT-INV-OPEN and eliminate the view shift in the precondition. The specification of the other functions can be proven similarly.

We now focus on showing that for each of the *ctape* definitions, they satisfy the presampling specification. For $ctape_1$, we see after unfolding its definition, the statement of the presampling specification is the same as that of PUPD-PRESAMPLE-EXP and hence holds directly. For $ctape_2$, it suffices to prove the following probabilistic update:

$$\begin{array}{l}(\mathbb{E}_{\mathfrak{U}_3}[\mathcal{F}] \le \varepsilon) \mathrel{-\!\!*} \kappa \hookrightarrow (1, \vec{n}) \mathrel{-\!\!*} \lightning(\varepsilon) \mathrel{-\!\!*} \\ \rightsquigarrow_{\mathcal{E}} \exists v_1, v_2.\ \lightning(\mathcal{F}(v_1 \cdot 2 + v_2)) * \kappa \hookrightarrow (1, \vec{n} \cdot [v_1, v_2])\end{array} \tag{7}$$

This probabilistic update is valid because we can do two presamples consecutively via PUPD-BIND. We first apply PUPD-PRESAMPLE-EXP to presample the first bit, choosing the first error splitting function $\mathcal{F}_a \triangleq \lambda\, b.\ \text{if } b = 1 \text{ then } \mathcal{F}(2) + \mathcal{F}(3) \text{ else } \mathcal{F}(0) + \mathcal{F}(1)$. We then do a case distinction on the bit that was sampled. If it is 0, we apply PUPD-PRESAMPLE-EXP again, choosing the error splitting function to be $\mathcal{F}_b \triangleq \lambda\, b.\ \text{if } b = 1 \text{ then } \mathcal{F}(1) \text{ else } \mathcal{F}(0)$. Otherwise, we choose $\mathcal{F}_b \triangleq \lambda\, b.\ \text{if } b = 1 \text{ then } \mathcal{F}(3) \text{ else } \mathcal{F}(2)$.

For $ctape_3$ we want to show that we can repeatedly presample enough values onto the tape such that the last element is smaller than 4 and all values beforehand are 4, while distributing the error credit according to the final value. This can be shown by the following lemma:

$$\begin{array}{l}(\mathbb{E}_{\mathfrak{U}_3}[\mathcal{F}] \le \varepsilon) \mathrel{-\!\!*} (\exists \vec{m}.\ \mathsf{filter}\ (\lambda\, x.\ x < 4)\ \vec{m} = \vec{n} * \kappa \hookrightarrow (4, \vec{m})) \mathrel{-\!\!*} \lightning(\varepsilon) \mathrel{-\!\!*} \\ \rightsquigarrow_{\mathcal{E}} \exists n.\ 0 \le n < 4 * \lightning(\mathcal{F}(n)) * \exists \vec{m}.\ \mathsf{filter}\ (\lambda\, x.\ x < 4)\ \vec{m} = (\vec{n} \cdot [n]) * \kappa \hookrightarrow (4, \vec{m})\end{array} \tag{8}$$

We prove this probabilistic update through *induction by error amplification*. We first apply PUPD-BIND and PUPD-ERR to obtain some positive error credit $\lightning(\varepsilon')$ to get the following:

$$\begin{array}{l}(\mathbb{E}_{\mathfrak{U}_3}[\mathcal{F}] \le \varepsilon) \mathrel{-\!\!*} \varepsilon' > 0 \mathrel{-\!\!*} \lightning(\varepsilon') \mathrel{-\!\!*} (\exists \vec{m}.\ \mathsf{filter}\ (\lambda\, x.\ x < 4)\ \vec{m} = \vec{n} * \kappa \hookrightarrow (4, \vec{m})) \mathrel{-\!\!*} \lightning(\varepsilon) \mathrel{-\!\!*} \\ \rightsquigarrow_{\mathcal{E}} \exists n.\ 0 \le n < 4 * \lightning(\mathcal{F}(n)) * (\exists \vec{m}.\ \mathsf{filter}\ (\lambda\, x.\ x < 4)\ \vec{m} = (\vec{n} \cdot [n]) * \kappa \hookrightarrow (4, \vec{m}))\end{array}$$

Now we apply the induction by error amplification rule below (see Eris [3] for more details):

$$\frac{\varepsilon_1 > 0 \qquad k > 1 \qquad \lightning(\varepsilon_1) \qquad \forall \varepsilon_2.\ (\lightning(k \cdot \varepsilon_2) \mathrel{-\!\!*} P) \mathrel{-\!\!*} \lightning(\varepsilon_2) \mathrel{-\!\!*} P}{P}\ \textsc{ind-err-amp}$$

Morally this states that in order to prove P we can assume it holds guarded by an amount of credits amplified by a factor k strictly greater than 1. We choose the amplification factor $k \triangleq 5$. It suffices to show (with the induction hypothesis highlighted):

$$(\mathbb{E}_{\mathfrak{U}3}[\mathcal{F}] \le \varepsilon) \mathrel{-\!\!*} \varepsilon' > 0 \mathrel{-\!\!*} \left(\lightning(5 \cdot \varepsilon') \mathrel{-\!\!*} (\exists \vec{m}.\ \mathsf{filter}\ (\lambda\, x.\, x < 4)\ \vec{m} = \vec{n} * \kappa \hookrightarrow (4, \vec{m})) \mathrel{-\!\!*} \ldots\right) \mathrel{-\!\!*}$$
$$\lightning(\varepsilon') \mathrel{-\!\!*} (\exists \vec{m}.\ \mathsf{filter}\ (\lambda\, x.\, x < 4)\ \vec{m} = \vec{n} * \kappa \hookrightarrow (4, \vec{m})) \mathrel{-\!\!*} \lightning(\varepsilon) \mathrel{-\!\!*} \ldots$$

We can now combine $\lightning(\varepsilon) * \lightning(\varepsilon')$ with ERR-SPLIT and apply PUPD-PRESAMPLE-EXP with $\lightning(\varepsilon + \varepsilon')$ as the initial error budget. We choose the distribution function to be $\lambda\, x.$ if $x < 4$ then $\mathcal{F}(x)$ else $\varepsilon{+}5{\cdot}\varepsilon'$. After a single presampling step, we do a case distinction on whether the presampled value is 4 or not. If it is, then we establish the conclusion with the induction hypothesis since we successfully amplified the error credit $\lightning(\varepsilon')$ by a factor of 5. Otherwise, we presampled an "accepted" value, and we can directly establish the goal via PUPD-RET.

6 Case Studies

In this section, we present several other case studies that we have verified using Coneris.

6.1 Thread-Safe Hash Functions

Hash functions are often assumed to behave *uniformly* [11]. That is, a hash function h from a set of keys K to a set of values V behaves as if, for each key k, the hash $h(k)$ is randomly sampled from a uniform distribution over V independently of all other keys. This assumption can be modeled using an idealized hash function that uses a mutable map, which serves as a cache of hashes computed so far [30]. If the key has already been hashed, we return the value stored in the map, otherwise we sample a fresh value uniformly, store it in the cache, and return it. In the concurrent setting, however, this does not suffice: if two threads concurrently attempt to hash the same key k, they may end up with different hash values. If one thread gets preempted by the scheduler right after sampling, a second thread could overtake and sample a different value before the first thread stores its value to the cache.

We implement a thread-safe idealized hash function using a lock. To hash a value, one first acquires the lock, then samples the key and stores it to the cache, before releasing the lock again. While the implementation is uninteresting, its specification is not. In particular, we give a specification that offers exclusive ownership of each key k and the ability to presample the hash $h(k)$. As we later see in §6.2, this ability can greatly simplify the probabilistic analysis of concurrent data structures that use hashing.

The hashInit function initializes a new hash function and satisfies the specification below.

$$\{\mathsf{True}\}\ \mathsf{hashInit}\ ()\ \{h.\ \exists \gamma.\ \mathsf{hashFun}\ \gamma\ h * \mathop{\ast}_{k \in K} \mathsf{hashKey}\ \gamma\ k\ -\}$$

Here, γ is a ghost name logically identifying the hash function. The abstract predicate $\mathsf{hashFun}\ \gamma\ h$ is duplicable, *i.e.*, $\mathsf{hashFun}\ \gamma\ h \dashv\vdash \mathsf{hashFun}\ \gamma\ h * \mathsf{hashFun}\ \gamma\ h$, while $\mathsf{hashKey}\ \gamma\ k\ -$ represents that key k has not yet been hashed, and is exclusive, *i.e.*, $\mathsf{hashKey}\ \gamma\ k\ - * \mathsf{hashKey}\ \gamma\ k - \vdash \mathsf{False}$. When invoking a hash function on a key with an undecided value, a fresh value $v \in V$ is sampled and $\mathsf{hashKey}\ \gamma\ k\ v$ is returned. The predicate $\mathsf{hashKey}\ \gamma\ k\ v$ is duplicable and each subsequent

```
bfInit () ≜
    let hfs = List.init k (λ _. hash_init ()) in
    let arr = Array.init S false in
    (hfs, arr)

bfAdd bfl x ≜
    let (hfs, arr) = bfl in
    List.Iter(λ h. let i = h x in
                   arr[i] ← true) hfs

bfLookup bfl y ≜
    let (hfs, arr) = bfl in
    let res = ref true in
    List.Iter(λ h. let i = h y in
                   res ← ! res && arr[i]) hfs;
    ! res

bfMain xs y ≜
    let bfl = bfInit () in
    (rec f zs =
      match zs with
      | [ ] ⇒ ()
      | z :: zs′ ⇒ (bfAdd bfl z) ||| (f zs′)
      end) ks;
    bfLookup bfl k
```

Fig. 8. Implementation of a concurrent Bloom filter.

invocation is guaranteed to return v.

$$\{\mathsf{hashFun}\ \gamma\ h * \mathsf{hashKey}\ \gamma\ k\ -\}\ h\ k\ \{v.\ \exists v \in V.\ \mathsf{hashKey}\ \gamma\ k\ v\}$$
$$\{\mathsf{hashFun}\ \gamma\ h * \mathsf{hashKey}\ \gamma\ k\ v\}\ h\ k\ \{w.\ w = v\}$$

However, hash values can also be presampled and error credits redistributed across the possible outcomes of the presampling using the probabilistic update below.

$$\mathsf{hashFun}\ \gamma\ f * \mathsf{hashKey}\ \gamma\ k\ -\ * \lightning(\varepsilon) \mathrel{-\!\!*}$$
$$\pmb{\upsilon}_{\top}\ \exists v \in V.\ \mathsf{hashKey}\ \gamma\ k\ v * (v \in X * \lightning(\varepsilon_1)) \vee (v \notin X * \lightning(\varepsilon_0))$$

Here $X \subseteq V$ is some set of hash values and $\varepsilon_1, \varepsilon_0 \in [0, 1]$ such that $\varepsilon_1 \cdot |X| + \varepsilon_0 \cdot (|V| - |X|) \leq \varepsilon \cdot |V|$. For example, by picking $\varepsilon_1 \triangleq 1$, $\varepsilon_0 \triangleq 0$, and $\varepsilon \triangleq |X|/|V|$ one can spend ε error credits to avoid the outcomes in X when determining the hash $h(k)$.

We show the specification by allocating a fresh presampling tape for each key in K. A similar idea is used in previous work [20] to show refinement of lazy and eager hash functions. In our specification, intuitively, $\mathsf{hashKey}\ \gamma\ k\ -$ denotes exclusive ownership of k's presampling tape which is transferred to an invariant after presampling. This invariant captures that, for all keys k, *either* n has been presampled onto k's tape *or* n has been stored at entry k in the hash function's cache.

6.2 Bloom Filter

Bloom filters are approximate data structures to represent sets, with operations for inserting elements and querying for membership. In their most basic, sequential presentation, a Bloom filter consists of an array of bits of a fixed size S, initially set to 0, and a list of hash functions $(h_1, \ldots, h_k)$ of some fixed length k. When inserting an element x, we compute $(h_1(x) \mod S, \ldots, h_k(x) \mod S)$ and set those indices to 1. When checking if an element y is in the set, we also compute $(h_1(y) \mod S, \ldots, h_k(y) \mod S)$, and look up those indices in the array. If they are all set to 1, we answer positively, otherwise we answer negatively. Thus, when checking the membership of an element that is not in the set there exists a small probability of observing a false positive if there are hash collisions with previously inserted elements. Computing this probability is challenging and requires involved combinatorial reasoning, in fact Bloom's original analysis [12] gave the wrong bound.

An efficient concurrent implementation of a Bloom filter allows parallel insertions, since concurrent writes to the same entry in the array would both set the entry to 1. In this case study, we implement a concurrent Bloom filter and prove a bound on the probability of observing a false positive result on a membership query. We use the concurrent hash module presented in §6.1 to implement this concurrent Bloom filter example (see Figure 8).

First consider N sequential insertions $x_1, \ldots, x_N$ followed by checking membership for some $y \notin \{x_1, \ldots, x_N\}$. From a mathematical perspective, the probability of false positive corresponds to the following experiment: first sample a batch of $k \cdot N$ integers uniformly at random in $\{0, \ldots, S-1\}$. Now sample a second batch of N integers in the same manner. What is the probability that they are *all* in the first batch? The exact bound was first calculated by Bose et al. [14], and in later work, Gopinathan and Sergey [19] mechanized the proof.

Now suppose that the insertions $x_1, \ldots, x_N$ happen in N parallel threads. Intuitively, concurrent implementations of Bloom filters should have the same probability of false positive, since parallel queries to hash functions are independent. Using our logic, we can make this intuition concrete, and prove that the bound in the concurrent setting indeed corresponds to the sequential one.

Our modular approach allows us to simplify the mathematical reasoning within the proof of the specification and defer all complex combinatiorial reasoning to the meta-level. The proof crucially relies on both the stateful representation of error probabilities (*i.e.*, error credits) as well as the notion of randomized logical atomicity, which allows us to presample all randomness in advance.

The key observation is that the probability of false positive follows a simple recurrence. Let $\mathsf{E}_{\mathsf{fp}}(l, b)$ be the probability of observing a false positive for a single membership query after setting l uniformly selected indices to 1 in an array that already contains b bits set to 1.

$$\mathsf{E}_{\mathsf{fp}}(0, b) \triangleq (\tfrac{b}{S})^k$$
$$\mathsf{E}_{\mathsf{fp}}(l+1, b) \triangleq \tfrac{b}{S} \cdot \mathsf{E}_{\mathsf{fp}}(l, b) + \tfrac{S-b}{S} \cdot \mathsf{E}_{\mathsf{fp}}(l, b+1)$$

Our analysis can therefore assume that every time we hash, we start with $\lightning\left(\mathsf{E}_{\mathsf{fp}}(l+1, b)\right)$ for some l, where b is the number of distinct hash outputs that have been observed so far, and then obtain either $\lightning\left(\mathsf{E}_{\mathsf{fp}}(l, b)\right)$ or $\lightning\left(\mathsf{E}_{\mathsf{fp}}(l, b+1)\right)$ depending on whether or not the output of the hash is a new one or not. This means that the decision on how to distribute credits can be done locally everytime we hash a new element.

With this in mind, we can prove the following spec:

$$\left\{\mathsf{NoDup}(\mathsf{xs}) * \mathsf{y} \notin \mathsf{xs} * \lightning\left(\mathsf{E}_{\mathsf{fp}}(\mathsf{k} \cdot |\mathsf{xs}|, 0)\right)\right\} \mathsf{bfMain}\ xs\ y\ \{v.\, v = \mathsf{false}\} \tag{9}$$

i.e., the probability of false positive is at most $\mathsf{E}_{\mathsf{fp}}(k \cdot |xs|, 0)$, which corresponds to the theoretical bound given by Bose et al. [14] for the sequential setting[5]. In order to simplify reasoning about concurrent hashing, we presample the hash outcomes for every key in xs in advance, using the hash specification in §6.1. It is at this point that most reasoning about probabilities takes place, and that we do the distribution of error credits. After this phase, we have $\lightning\left(\mathsf{E}_{\mathsf{fp}}(0, B)\right)$ for some B, as well as predicates of the form $\mathsf{hashKey}\ \gamma\ k_i\ v_i$ for every key and every hash, and we know that the set of presampled hash outcomes has cardinality B. Then we execute all insertions, with an invariant that ensures that the array never has more than B elements set to 1. Finally, we can do a lookup, and use our error credits $\lightning\left(\mathsf{E}_{\mathsf{fp}}(0, B)\right)$ to ensure that at least one of the indices we look up is set to 0, which guarantees that the query returns false.

[5]Note that their bound is given as a closed mathematical expression and we have not mechanized that it corresponds to our recursive definition.

To the best of our knowledge, we are the first to prove a tight bound on the probability of false positives for a concurrent Bloom filter (9). For more details on the analysis, we refer the reader to our Rocq development.

6.3 Lazy Random Sampler

In this section, we consider the implementation of a concurrent lazy one-shot random sampler. This sampler is lazy in the sense that we only perform the sampling the first time the thunk is invoked and we store the result in a reference that is read from whenever the thunk is invoked again.

An excerpt of the implementation of the lazy random sampler is shown in Figure 9. The function *lazyRandInit* creates a tuple containing a lock and a reference that points to None. When we call *lazyRandf* with the tuple and a tape label, we acquire the lock and load the value of what the location is pointing to. If it is Some x, we directly return x. Otherwise, we sample x from the tape with the function *randf* κ from some Rand module and store it into the location. Here the Rand module is some abstract module that samples $\{0, \ldots, N\}$ uniformly from some abstract tape κ, where N is some parameter fixed in advance, i.e. *randf* κ acts like a normal rand N κ (we provide more details in Appendix C.1). We additionally take in an extra argument tid in *lazyRandf* and store it into the location together with the sampled value to also track the first thread id that succeeds in acquiring the lock and performing the actual randomized operation. We release the lock right before we return from the function.

```
lazyRandInit ≜ λ _.
    let ℓ = ref None in
    let lo = newLock () in
    (lo, ℓ)

lazyRandf ≜ λ (lo, ℓ), κ, tid.
    acquire lo;
    let v = match ! ℓ with
      | Some x ⇒ x
      | None ⇒
        let x = (randf κ, tid) in
        ℓ ← Some x; x
    end in
    release lo; v
```

Fig. 9. Implementation of a lazy random sampler.

To motivate the specification of this lazy random sampler module, consider a client program *lazyRace* that uses the module. In this example, we set the parameter of the internal Rand module to be 1, so *randf* samples uniformly between 0 and 1. (The function *lazyAllocTape* in this example creates a tape for this lazy random sampler, and we omit the code for brevity.)

```
lazyRace ≜ let r = lazyRandInit () in
    (lazyRandf r (lazyAllocTape ()) 0) ||| (lazyRandf r (lazyAllocTape ()) 1)
```

In the *lazyRace* program, we create a lazy random sampler and fork two threads. Each thread attempts to sample from it but they pass a different tid as the thread id argument. It should be the case that both threads return the same tuple value $x = (x_1, x_2)$; intuitively, regardless of how the threads are scheduled, the thread that is executed last must read the value stored by the the thread that is executed first. Consider the following specification of *lazyRace* where for both return values of the threads, we have $x_1 = x_2$ with error probability $1/2$.

$$\{\lightning\,(1/2)\}\ \mathit{lazyRace}\ \{v.\exists n.\, v = ((n, n), (n, n))\}$$

This is true morally because whichever threads gets scheduled first to perform the sampling, we can use $↯(1/2)$ to avoid sampling a value from *randf* that is different from the *tid* passed, ensuring that the sampled value is identical to the *tid*. However, there is some subtlety in the proof of this Hoare triple. In particular, we cannot perform any presampling in advance of the actual *lazyRandf*. If we directly attempt to presample a value to each tape on both threads (avoiding the corresponding *tid*), we need to pay up to $↯(3/4)$ error credits because we are doing two presampling calls, where ideally, we should only need to do one. One might try to rewrite *lazyRace* such that both threads share the same tape, but this does not solve the presampling problem directly. In particular, before either threads call *lazyRandf*, we do not know what value to sample onto the tape. Whatever value is presampled, the scheduler can deliberately choose to schedule the threads in a way such that the *tid* of the winning thread does not match the presampled value. In other words, we want to delay the operation of presampling and perform it not before the *lazyRandf* call, but during it.

Given this observation, the specification of the lazy random sampler module is written in a way that allows presampling to be performed within the *lazyRandf* call dynamically. We show the specification for presampling and *lazyRandf* in Figure 10.

$$(\mathbb{E}_{\mathfrak{U}N}[\mathcal{F}] \le \varepsilon) \twoheadrightarrow \mathit{isLazyRand}\ \mathit{lr}\ P\ \iota\ \gamma \twoheadrightarrow \mathit{lazyTape}\ \kappa\ \mathsf{None}\ \gamma \twoheadrightarrow ↯(\varepsilon) \twoheadrightarrow$$
$$\rightsquigarrow_{\mathcal{E}\uplus\{\iota\}} \exists n.\ ↯(\mathcal{F}(n)) \ *\ \mathit{lazyTape}\ \kappa\ (\mathsf{Some}\, n)\ \gamma$$

(a) Presampling specification.

$$\mathrm{R}\ n \triangleq \begin{cases} P\ n * \mathit{lazyAuth}\ n\ \gamma * Q_1\ x\ y & \text{if } n = \mathsf{Some}(x,y) \\ \exists n'.\ \mathit{lazyTape}\ \kappa\ (\mathsf{Some}\, n')\ \gamma * (\mathit{lazyTape}\ \kappa\ \mathsf{None}\ \gamma \twoheadrightarrow & \text{if } n = \mathsf{None} \\ \quad \Rrightarrow_{\top} P\ (n', \mathit{tid}) * \mathit{lazyAuth}\ (n', \mathit{tid})\ \gamma * Q_2\ n'\ \mathit{tid}) & \end{cases}$$

$$\{\mathit{isLazyRand}\ \mathit{lr}\ P\ \iota\ \gamma * \ (\forall n.\ P\ n \twoheadrightarrow \mathit{lazyAuth}\ n\ \gamma \twoheadrightarrow \rightsquigarrow_{\top} R\ n)\}$$
$$\quad \mathit{lazyRandf}\ \mathit{lr}\ \kappa\ \mathit{tid}$$
$$\{(x,y).\ Q_1\ x\ y \vee Q_2\ x\ y\}_{\mathcal{E}}$$

(b) Specification of *lazyRandf*.

Fig. 10. Excerpt of the specification of the lazy random sampler module.

The presampling specification for the lazy random sampler is not too different from the other previous examples; the main difference is that the abstract tapes for the module *lazyTape* stores an option type instead of a list. Since for each tape, only the first value could ever be relevant in that it is chosen to be the value stored in the reference, there is no reason to presample more than one value into a single tape.

Now, let us focus on the more complicated specification for the *lazyRandf* function. Firstly, notice that the lazy random sampler predicate *isLazyRand* takes in an additional predicate P as an argument. Intuitively, P is the invariant protected by the lock, and if the reference maps to the value n, it is the case that $P\ n$ holds whenever we access the lock and release it. The precondition of the *lazyRandf* function requires two resources. The first being the abstract predicate *isLazyRand* and the second being a view shift. The view shift encodes how the state of the module changes throughout the call. The view shift starts by assuming that we have $P\ n$ and $\mathit{lazyAuth}\ n\ \gamma$ for some n, which represents the operation of acquiring the lock and gaining access to the authoritative

state of the lazy random sampler. We then perform a case distinction on n. If it is $\mathsf{Some}(x, y)$, then this means that the lazy random sampler has already committed to a value, so we return directly by releasing the lock and establishing some postcondition $Q_1\ x\ y$. Otherwise, if it is None, we reach the branch where we have to do a randomized sampling. Here we are allowed to perform some probabilistic update operation to provide a non-empty *lazyTape* (since the view shift is implemented with a $\leadsto_\top$), and it suffices to prove that after reading that value n' in the tape, we establish the authoritative part of the lazy random sampler with the reference storing (n', tid) and some postcondition $Q_2\ n'\ tid$. If all preconditions hold, then the return value of the function is some pair (x, y), where either $Q_1\ x\ y$ or $Q_2\ x\ y$ holds.

The key ingenuity of the specification of *lazyRandf* is that the view shift is described by the $\leadsto$ modality, instead of the regular fancy update modality $\Rightarrow$, allowing us to perform presampling on abstract tapes *within* the function call in addition to outside of it. In particular, we can choose to perform a presampling action on a tape or not depending on whether the sampler is storing a None (it has not been invoked before) or not, which we know after a case distinction on the value of n after gaining access to the lock. This flexibility allows us to prove the specification of the *lazyRace*, by only performing a *single* presampling within the first invocation of the function call *lazyRandf*.

6.4 Other Case Studies

Other case studies demonstrating the flexibility of our approach in verifying concurrent randomized data structures can be found in Appendix C. We define a Rand module that captures the operation of sampling from a uniform distribution atomically and we provide three implementations that satisfy it (similar to the implementations in the randomized counter module from §5.3). This is the abstract Rand module used to implement the lazy random sampler in §6.3. We also implement a concurrent collision-free hash data structure and show that it meets an amortized specification where the error required for each operation is amortized across a fixed number of insertions.

7 Semantic Model and Soundness

Coneris is implemented on top of the Iris [25] base logic, which in isolation, is simply a higher-order separation logic not tied to any specific programming language. In this section, we define the semantic model of Coneris and explain how to prove the soundness of the program logic.

7.1 Model

Weakest Precondition. The Coneris Hoare triple is defined in terms of a weakest precondition predicate as follows:

$$\{P\}\ e\ \{Q\} \triangleq \Box(P \mathrel{-\!\!*} \mathsf{wp}\ e\ \{Q\})$$

Expressing Hoare triples in term of a weakest precondition is standard for defining program logics [25], especially for other similar Iris non-probabilistic logics. The definition of the weakest precondition is however novel, which we detail below. Note that the weakest precondition is defined as a guarded fixed point: the recursive occurrences of the weakest precondition appear under the later modality $\triangleright$ on the last line.

$$\begin{aligned}\mathsf{wp}\ e_1\ \{\Phi\} \triangleq \forall \sigma_1, \varepsilon_1.\ S\ \sigma_1\ \varepsilon_1 \mathrel{-\!\!*} {}_\top\!\Rightarrow_\emptyset \mathsf{sstep}\ \sigma_1\ \varepsilon_1\ \{\sigma_2, \varepsilon_2.\\ \big(e_1 \in \mathit{Val} * {}_\emptyset\!\Rightarrow_\top S\ \sigma_2\ \varepsilon_2 * \Phi\ e_1\big) \vee\\ \big(e_1 \notin \mathit{Val} * \mathsf{pstep}\ (e_1, \sigma_2)\ \varepsilon_2\ \{e_2, \sigma_3, l, \varepsilon_3.\\ \triangleright\, \mathsf{sstep}\ \sigma_3\ \varepsilon_3\ \{\sigma_4, \varepsilon_4.\ {}_\emptyset\!\Rightarrow_\top S\ \sigma_4\ \varepsilon_4 * \mathsf{wp}\ e_2\ \{\Phi\} * \mathop{\Large *}_{e' \in l} \mathsf{wp}\ e'\ \{\mathsf{True}\}\}\big)\}\end{aligned}$$

One can intuitively understand $\mathsf{wp}\ e\ \{\Phi\}$ as a proposition that describes that e is *safe*, meaning it does not get stuck, and that for every possible return value v, the postcondition $\Phi\ v$ holds.

We now explain the definition of the weakest precondition in detail. At the beginning, we assume the ownership of a *state interpretation* $S\ \sigma_1\ err_1$ for some state σ_1 and error value ε_1. This state interpretation $S : State \to \mathbb{R}_{\geq 0} \to iProp$ gives meaning to the ownership of references $\ell \mapsto v$, tapes $\kappa \hookrightarrow (N, \vec{n})$, and error credits $\pounds(\varepsilon)$. The resource algebra used to instantiate the state interpretation is standard, and we refer the readers to Aguirre et al. [3] for more details.

After that, we perform a view shift through the update modality ${}_{\top}\Rrightarrow_{\emptyset}$, which intuitively means we open all invariants temporarily and that we have access to the resources of all invariants we defined previously. Following the view shift, we need to prove a *state step precondition* $\mathsf{sstep}\ \sigma_1\ \varepsilon_1\{\dots\}$. The exact definition of the state step precondition is explained later. For now, we can think of the state step precondition as the modality that allows instantaneous probability-preserving updates supported by the probabilistic update modality $\leadsto P$. Given state σ_1 and error budget ε_1, we can perform any number of probability-preserving updates to step to resulting state σ_2 and leftover error budget ε_2, which must satisfy the rest of the continuation.

The next part of the weakest precondition depends on a case split on the expression e_1. In the first case, where e_1 is a value, we do a view shift ${}_{\emptyset}\Rrightarrow_{\top}$ where we re-establish *all* invariants, return the updated state interpretation and show that the return value e_1 satisfies the postcondition Φ. Otherwise, if e_1 is not a value, we have to prove a program step precondition $\mathsf{pstep}\ (e_1, \sigma_2)\ \varepsilon_2\ \{\dots\}$. We later explain the specifics of this precondition modality, but for now, one can loosely understand the connective as somewhat similar to the state step precondition, where we take an actual step on the configuration (e_1, σ_2), instead of performing a probabilistic update on σ_2. After the single step to resulting expression e_2, state σ_3, forking the list of expressions l with leftover error budget ε_3, we prove another state step precondition, which we can ignore here.[6] Finally, we re-establish all invariants after the view shift ${}_{\emptyset}\Rrightarrow_{\top}$, return the state interpretation, show that $\mathsf{wp}\ e_2\ \{\Phi\}$ holds, and $\mathsf{wp}\ e'\ \{\mathsf{True}\}$ holds for all e' in the forked list l.

$$\frac{1 \leq \varepsilon}{\mathsf{sstep}\ \sigma\ \varepsilon\ \{\Phi\}}\ \textsc{state-step-err-1} \qquad \frac{\Phi(\sigma, \varepsilon)}{\mathsf{sstep}\ \sigma\ \varepsilon\ \{\Phi\}}\ \textsc{state-step-ret} \qquad \frac{\forall \varepsilon'.\ \varepsilon < \varepsilon' \twoheadrightarrow \mathsf{sstep}\ \sigma\ \varepsilon'\ \{\Phi\}}{\mathsf{sstep}\ \sigma\ \varepsilon\ \{\Phi\}}\ \textsc{state-step-continuous}$$

$$\frac{\mathbb{E}_\mu[\mathcal{F}] \leq \varepsilon \qquad \mathsf{schErasable}(\mu, \sigma_1) \qquad \forall \sigma_2.\ 0 < \mu(\sigma_2) \twoheadrightarrow \mathsf{sstep}\ \sigma_2\ (\mathcal{F}(\sigma_2))\ \{\Phi\}}{\mathsf{sstep}\ \sigma_1\ \varepsilon\ \{\Phi\}}\ \textsc{state-step-exp}$$

Fig. 11. Inductive Definition of the State Step Precondition $\mathsf{sstep}\ \sigma\ \varepsilon\ \{\Phi\}$.

State and Program Step Preconditions. The state step precondition is defined inductively by four inference rules presented in Figure 11. Firstly, if the error budget ε is larger or equal to 1, the precondition holds trivially as all sub-distributions have mass smaller or equal to 1 (STATE-STEP-ERR-1). If the predicate Φ holds for the current state and error budget, the precondition also holds (STATE-STEP-RET). The third rule STATE-STEP-CONTINUOUS states that the precondition holds if for error budget ε' larger than the ε (in particular, ε' can be arbitrarily close to ε), then the precondition does in fact hold for ε as well. This is the main rule that allows us to create error credits from thin air (PUPD-ERR), letting us exploit the fact that the real numbers are complete at the level of Coneris.

Lastly, STATE-STEP-EXP is the main interesting rule, which relies on the following auxiliary definition.

[6] This extra state step precondition is only used to validate certain invariant opening properties not discussed in this paper.

Definition 7.1. *A distribution on states μ is a* scheduler erasable state update *of $\sigma \in State$, written as* $\mathsf{schErasable}(\mu, \sigma)$*, if for all schedulers ζ, scheduler states Ξ, thread pools $\vec{e}$, and any number of execution steps n, we have*

$$(\mu \gg= (\lambda\, \sigma'.\, \mathsf{pexec}_{\zeta,n}(\Xi, (\vec{e}, \sigma')))).\mathsf{tp} \;=\; (\mathsf{pexec}_{\zeta,n}(\Xi, (\vec{e}, \sigma))).\mathsf{tp}$$

where we write $-.\mathsf{tp}$ for the function that projects out the thread pool component from a distribution on configurations.

A distribution μ is thus a scheduler erasable state update of σ if the probability of executing $\vec{e}$ from state σ to any particular list of threads is the same if we first update the state with respect to μ and then execute $\vec{e}$. Recall that the operational semantics requires schedulers to be invariant under changes to presampling tapes; such changes thus constitute scheduler erasable state updates.

The STATE-STEP-EXP rule then states that if we can find a function $\mathcal{F} : State \to [0,1]$ and a distribution $\mu : \mathcal{D}(State)$ such that (1) the expectation of $\mathcal{F}$ with respect to μ is at most ε, (2) μ is scheduler erasable with respect to σ_1, and (3) for all σ_2, the continuation $\mathsf{sstep}\ \sigma_2\ (\mathcal{F}(\sigma_2))\ \{\Phi\}$ holds, then $\mathsf{sstep}\ \sigma_1\ \varepsilon\ \{\Phi\}$ holds. This is the rule that allows us to do presampling on tapes, since the presampling action is a scheduler erasable operation.

The program step precondition is defined by a single inference rule PROG-STEP-EXP. It is similar to that of STATE-STEP-EXP, except that we take exactly one step of the configuration (e_1, σ_1). In detail, $\mathsf{pstep}\ (e_1, \sigma_1)\ \varepsilon\ \{\Phi\}$ holds if the configuration (e_1, σ_1) is reducible and there exists some function $\mathcal{F} : Expr \times State \times List(Expr) \to [0,1]$ whose expectation with respect to $\mathsf{step}(e_1, \sigma_1)$ is smaller or equal to ε, and for all (e_2, σ_2, l), the continuation $\Phi(e_2, \sigma_2, l, \mathcal{F}(e_2, \sigma_2, l))$ holds.

PROG-STEP-EXP

$$\frac{\mathsf{red}(e_1, \sigma_1) \qquad \mathbb{E}_{\mathsf{step}(e_1,\sigma_1)}[\mathcal{F}] \le \varepsilon \qquad \forall (e_2, \sigma_2, l).\ 0 < \mathsf{step}(e_1, \sigma_1)(e_2, \sigma_2, l) \twoheadrightarrow \Phi(e_2, \sigma_2, l, \mathcal{F}(e_2, \sigma_2, l))}{\mathsf{pstep}\ (e_1, \sigma_1)\ \varepsilon\ \{\Phi\}}$$

Probabilistic Update Modality. Recall that the probabilistic update modality ${}_{\mathcal{E}_1}\!\rightsquigarrow_{\mathcal{E}_2} P$ depends on two masks $\mathcal{E}_1$ and $\mathcal{E}_2$. These extra mask parameters are used to track the opening of invariants and prevent us from opening the same invariant twice (which is unsound). One can understand ${}_{\mathcal{E}_1}\!\rightsquigarrow_{\mathcal{E}_2} P$ as that we can perform a probability-preserving update to get the resources P with the possibility of accessing resources of invariants in the mask $\mathcal{E}_1$ and reestablishing resources of invariants in the mask $\mathcal{E}_2$ in the end.

$${}_{\mathcal{E}_1}\!\rightsquigarrow_{\mathcal{E}_2} P \triangleq \forall \sigma_1, \varepsilon_1.\, S\ \sigma_1\ \varepsilon_1 \;\twoheadrightarrow\; {}_{\mathcal{E}_1}\!\Rrightarrow_{\emptyset} \mathsf{sstep}\ \sigma_1\ \varepsilon_1\ \{\sigma_2, \varepsilon_2.\ {}_{\emptyset}\!\Rrightarrow_{\mathcal{E}_2} S\ \sigma_2\ \varepsilon_2 * P\}$$

The definition of the probabilistic update modality resembles a simplified version of the weakest precondition, where we only perform a single state step. Specifically, ${}_{\mathcal{E}_1}\!\rightsquigarrow_{\mathcal{E}_2} P$ holds if after assuming some state interpretation $S\ \sigma_1\ \varepsilon_1$, we can open all invariants in $\mathcal{E}_1$ through the view shift ${}_{\mathcal{E}_1}\!\Rrightarrow_{\emptyset}$, and prove a state step precondition with the input parameters σ_1 and ε_1. Given resulting state σ_2 and error budget ε_2 after the state step precondition, we re-establish all invariants in the mask $\mathcal{E}_2$ with the view shift ${}_{\emptyset}\!\Rrightarrow_{\mathcal{E}_2}$ and give back the state interpretation and prove the resource P.

7.2 Soundness

The soundness of Coneris comes in two flavours, the correctness adequacy theorem Theorem 4.1 and the safety theorem Theorem 4.2. We now briefly describe the overall structure proof of the correctness adequacy theorem; the proof of the safety theorem is similar and is omitted.

We first prove an intermediate lemma:

Lemma 7.2. *If* $\lightning(\varepsilon) \vdash \mathsf{wp}\ e\ \{\phi\} * \ast_{e' \in \vec{e}'} \mathsf{wp}\ e'\ \{\mathsf{True}\}$*, then for all schedulers* ζ*, states* σ*, and natural numbers* n*,* $\Pr_{\mathsf{exec}_{\zeta,n}(e \cdot \vec{e}', \sigma)}[\neg\phi] \leq \varepsilon$.

This lemma is proven by induction on n and structural induction on the state step precondition fixed point. For each step, we unfold the definition of exec to determine which thread the scheduler chooses to step next. We unfold the definition of the corresponding weakest precondition proposition (the one that matches the thread chosen to step), and show that the sstep and pstep modalities satisfies monadic composition, allowing us to compose the errors.

By taking $\vec{e}'$ to be the empty list of threads in Lemma 7.2, and taking the limit of n, we can then show $\Pr_{\mathsf{exec}_\zeta e}[\neg\phi] \leq \varepsilon$, which is the goal of the adequacy theorem.

8 Related Work

Approximate Reasoning. There are various approaches for tracking error probabilities in probabilistic programs. Approximate Hoare logic [6] uses a grading on Hoare triples to approximate error probabilities. Expectation-based logics such as that of Batz et al. [10], Morgan et al. [31] are defined with a weakest-precondition-style quantitative predicate transformer that computes the expected value of a program's postcondition, which can be used to derive approximate correctness bounds. Compared to our work, these logics are usually restricted to sequential, first-order imperative programs. Our method of using error credits to track error bounds is first used in Eris [3] to prove error bounds of sequential higher-order probabilistic programs.

Various other logics also considered reasoning about approximate correctness in the relational setting. apRHL [5, 9] relates the probability distribution of two programs through approximate probabilistic couplings, which can then be used to prove differential privacy. Inspired by Eris, error credits are used in Approxis [21] to prove approximate equivalences of higher-order programs.

Concurrent Probabilistic Program Logics. One of the first program logics developed for concurrent probabilistic programs is the probabilistic rely-guarantee calculus [29] (that extends the rely-guarantee logic [24]) that verifies the quantitative correctness of a probabilistic concurrent programs without local state. Later, Concurrent Quantitative Separation Logic [17] extends Quantitative Separation Logic [10] to reason about the lower bounds of probability to realize the postcondition of concurrent, heap-manipulating, randomized imperative programs. Compared to our work, it cannot establish strict error bounds that arise between the interleaving of threads (see the *conTwoAdd* example in §2) and cannot reason about programs in a (procedure-)modular way.

Polaris [35] is a logic for establishing refinements between concurrent probabilistic programs and a monadic representation via probabilistic couplings inspired by pRHL [4, 7, 8]. The simpler monadic model can then be studied to derive properties of the original programs, such as bounds on its expected value. The language considered by Coneris is inspired by that of Polaris; the syntax is the same, but in Coneris, we allow schedulers to be probabilistic as well. Compared to Coneris, Polaris is not as modular in the sense that it does not demonstrate how to compose refinements of different data structures. It also does not develop an approach for reasoning about logical atomicity.

Lohse and Garg [28] develop ExpIris, a variant of Iris that supports establishing bounds on the expected cost of concurrent higher-order programs with mutable state. In ExpIris, an upper bound budget on the number of steps a program can take is written as an additional parameter of weakest preconditions, called a *potential*. On randomized steps, this potential can be updated in an expectation-preserving way, similar to HT-RAND-EXP. However, because potentials are a parameter of the weakest precondition, instead of a separation logic resource like error credits, it is not possible to share them in an invariant, as we saw was necessary for obtaining tight analyzes in §2. ExpIris also does not provide any facilities to encode the notion of randomized logical atomicity, which we show is essential to reason about concurrent programs modularly. Although ExpIris

provides rules for reasoning about concurrency, there are no case studies provided that utilize concurrent constructs (*e.g.*, the fork construct).

Recently, Probabilistic Concurrent Outcome Logic [38] extends Demonic Outcome Logic [37] to reason about the distributions of outcomes from concurrent probabilistic programs. Although this logic is able to prove other probabilistic properties beyond the scope of Coneris, such as independence and conditioning, the programs considered are restricted to those without dynamically allocated state or higher-order functions, and the logic does not support defining ghost state.

Internalization of Linearizability. There is a long line of research on internalizing linearizability as a reasoning principle within concurrent program logic specifications. Jacobs and Piessens [23] first extended the resource-invariants-based method from Owicki and Gries [32] allowing users to parameterize the specification of concurrent functions with ghost code. Later, Svendsen et al. [34] further extended their idea and proposed a new style of specification using higher-order concurrent abstract predicates (HOCAP), building on top of CAP [16]. da Rocha Pinto et al. [15] introduced a different logic called TaDa, which proposed the use of atomic triples to capture logical atomicity of programs. There has also been much research in encoding logically atomic specifications within the Iris separation logic [26, 27]. In Coneris, we take inspiration from these logics, especially HOCAP, to capture randomized logical atomicity within *probabilistic* concurrent programs.

9 Conclusion

We presented Coneris, the first concurrent and probabilistic higher-order separation logic for error bound reasoning. Coneris captures randomized logical atomicity through the novel probabilistic update modality, enabling modular verification of concurrent programs that is out-of-scope for previous techniques. We demonstrated the flexibility of Coneris by verifying various examples modularly, most of which involve local state and intricate reasoning over randomness that arise from concurrency.

There are various directions for extending Coneris. Firstly, we would like to extend Coneris to enable verifying strict error bounds of concurrent probabilistic programs under restricted schedulers, such as those that cannot view the configuration of the program. It is also interesting to explore whether ideas from Approxis [21] can be used to extend Coneris into the relational setting to establish approximate bounds between concurrent probabilistic programs. Lastly, we would like to consider integrating cost credits from Tachis [22] into Coneris to reason about both the expected work and span time costs of concurrent probabilistic programs.

Acknowledgments

The first author would like to thank Amin Timany for enlightening discussions regarding HOCAP-style specifications. The authors also thank François Pottier for finding an error in an earlier description of the example in §2. This work was supported in part by the National Science Foundation, grant no. 2338317, the Carlsberg Foundation, grant no. CF23-0791, a Villum Investigator grant, no. 25804, Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation, and the European Union (ERC, CHORDS, 101096090). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

References

[1] M. Abadi and L. Lamport. 1988. The existence of refinement mappings. In *[1988] Proceedings. Third Annual Symposium on Logic in Computer Science*. 165–175. https://doi.org/10.1109/LICS.1988.5115

[2] Martín Abadi and Leslie Lamport. 1991. The existence of refinement mappings. *Theoretical Computer Science* 82, 2 (1991), 253–284. https://doi.org/10.1016/0304-3975(91)90224-P

[3] Alejandro Aguirre, Philipp G. Haselwarter, Markus de Medeiros, Kwing Hei Li, Simon Oddershede Gregersen, Joseph Tassarotti, and Lars Birkedal. 2024. Error Credits: Resourceful Reasoning about Error Bounds for Higher-Order Probabilistic Programs. *Proc. ACM Program. Lang.* 8, ICFP, Article 246 (Aug. 2024), 33 pages. https://doi.org/10.1145/3674635

[4] Gilles Barthe, Thomas Espitau, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2017. Proving uniformity and independence by self-composition and coupling. In *LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning (EPiC Series in Computing, Vol. 46)*, Thomas Eiter and David Sands (Eds.). EasyChair, 385–403. https://doi.org/10.29007/vz48

[5] Gilles Barthe, Noémie Fong, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2016. Advanced Probabilistic Couplings for Differential Privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 55–67. https://doi.org/10.1145/2976749.2978391

[6] Gilles Barthe, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2016. A Program Logic for Union Bounds. In *43rd International Colloquium on Automata, Languages, and Programming (ICALP 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 55)*, Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 107:1–107:15. https://doi.org/10.4230/LIPIcs.ICALP.2016.107

[7] Gilles Barthe, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2017. Coupling proofs are probabilistic product programs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 161–174. https://doi.org/10.1145/3009837.3009896

[8] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. 2012. Probabilistic Relational Hoare Logics for Computer-Aided Security Proofs. In *Mathematics of Program Construction*, Jeremy Gibbons and Pablo Nogueira (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–6.

[9] Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella-Béguelin. 2013. Probabilistic Relational Reasoning for Differential Privacy. *ACM Trans. Program. Lang. Syst.* 35, 3, Article 9 (Nov. 2013), 49 pages. https://doi.org/10.1145/2492061

[10] 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 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290347

[11] Mihir Bellare and Phillip Rogaway. 1993. Random oracles are practical: a paradigm for designing efficient protocols. In *Proceedings of the 1st ACM Conference on Computer and Communications Security* (Fairfax, Virginia, USA) *(CCS '93)*. Association for Computing Machinery, New York, NY, USA, 62–73. https://doi.org/10.1145/168588.168596

[12] Burton H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (1970), 422–426. https://doi.org/10.1145/362686.362692

[13] Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. 2013. Coquelicot: A User-Friendly Library of Real Analysis for Coq. (Sept. 2013). https://inria.hal.science/hal-00860648 working paper or preprint.

[14] Prosenjit Bose, Hua Guo, Evangelos Kranakis, Anil Maheshwari, Pat Morin, Jason Morrison, Michiel H. M. Smid, and Yihui Tang. 2008. On the false-positive rate of Bloom filters. *Inf. Process. Lett.* 108, 4 (2008), 210–213. https://doi.org/10.1016/J.IPL.2008.05.018

[15] Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A Logic for Time and Data Abstraction. In *ECOOP 2014 – Object-Oriented Programming*, Richard Jones (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 207–231.

[16] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. 2010. Concurrent Abstract Predicates. In *ECOOP 2010 – Object-Oriented Programming*, Theo D'Hondt (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 504–528.

[17] Ira Fesefeldt, Joost-Pieter Katoen, and Thomas Noll. 2022. Towards Concurrent Quantitative Separation Logic. In *33rd International Conference on Concurrency Theory (CONCUR 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 243)*, Bartek Klin, Sławomir Lasota, and Anca Muscholl (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 25:1–25:24. https://doi.org/10.4230/LIPIcs.CONCUR.2022.25

[18] Wojciech M. Golab, Lisa Higham, and Philipp Woelfel. 2011. Linearizable implementations do not suffice for randomized distributed computation. In *Proceedings of the 43rd ACM Symposium on Theory of Computing, STOC 2011, San Jose, CA, USA, 6-8 June 2011*, Lance Fortnow and Salil P. Vadhan (Eds.). ACM, 373–382. https://doi.org/10.1145/1993636.1993687

[19] Kiran Gopinathan and Ilya Sergey. 2020. Certifying Certainty and Uncertainty in Approximate Membership Query Structures. In *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24,*

2020, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12225), Shuvendu K. Lahiri and Chao Wang (Eds.). Springer, 279–303. https://doi.org/10.1007/978-3-030-53291-8_16

[20] Simon Oddershede Gregersen, Alejandro Aguirre, Philipp G. Haselwarter, Joseph Tassarotti, and Lars Birkedal. 2024. Asynchronous Probabilistic Couplings in Higher-Order Separation Logic. *Proc. ACM Program. Lang.* 8, POPL, Article 26 (Jan. 2024), 32 pages. https://doi.org/10.1145/3632868

[21] Philipp G. Haselwarter, Kwing Hei Li, Alejandro Aguirre, Simon Oddershede Gregersen, Joseph Tassarotti, and Lars Birkedal. 2025. Approximate Relational Reasoning for Higher-Order Probabilistic Programs. *Proc. ACM Program. Lang.* 9, POPL, Article 41 (Jan. 2025), 31 pages. https://doi.org/10.1145/3704877

[22] Philipp G. Haselwarter, Kwing Hei Li, Markus de Medeiros, Simon Oddershede Gregersen, Alejandro Aguirre, Joseph Tassarotti, and Lars Birkedal. 2024. Tachis: Higher-Order Separation Logic with Credits for Expected Costs. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 313 (Oct. 2024), 30 pages. https://doi.org/10.1145/3689753

[23] Bart Jacobs and Frank Piessens. 2011. Expressive modular fine-grained concurrency specification. *SIGPLAN Not.* 46, 1 (Jan. 2011), 271–282. https://doi.org/10.1145/1925844.1926417

[24] Cliff Jones. 1983. Specification and Design of (Parallel) Programs. *Proceedings Of Ifip Congress '* 83, 321–332.

[25] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

[26] Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. 2019. The future is ours: prophecy variables in separation logic. *Proc. ACM Program. Lang.* 4, POPL, Article 45 (Dec. 2019), 32 pages. https://doi.org/10.1145/3371113

[27] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. 637–650. https://doi.org/10.1145/2676726.2676980

[28] Janine Lohse and Deepak Garg. 2024. An Iris for Expected Cost Analysis. arXiv:2406.00884 [cs.PL]

[29] Annabelle McIver, Tahiry Rabehaja, and Georg Struth. 2016. Probabilistic rely-guarantee calculus. *Theoretical Computer Science* 655 (2016), 120–134. https://doi.org/10.1016/j.tcs.2016.01.016 Quantitative Aspects of Programming Languages and Systems (2013-14).

[30] Arno Mittelbach and Marc Fischlin. 2021. *The Theory of Hash Functions and Random Oracles - An Approach to Modern Cryptography*. Springer. https://doi.org/10.1007/978-3-030-63287-8

[31] Carroll Morgan, Annabelle McIver, and Karen Seidel. 1996. Probabilistic predicate transformers. *ACM Trans. Program. Lang. Syst.* 18, 3 (May 1996), 325–353. https://doi.org/10.1145/229542.229547

[32] Susan Owicki and David Gries. 1976. Verifying properties of parallel programs: an axiomatic approach. *Commun. ACM* 19, 5 (May 1976), 279–285. https://doi.org/10.1145/360051.360224

[33] Peter W. O'Hearn. 2007. Resources, concurrency, and local reasoning. *Theoretical Computer Science* 375, 1 (2007), 271–307. https://doi.org/10.1016/j.tcs.2006.12.035 Festschrift for John C. Reynolds's 70th birthday.

[34] Kasper Svendsen, Lars Birkedal, and Matthew Parkinson. 2013. Modular Reasoning about Separation of Concurrent Data Structures. In *Programming Languages and Systems*, Matthias Felleisen and Philippa Gardner (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 169–188.

[35] Joseph Tassarotti and Robert Harper. 2019. A separation logic for concurrent randomized programs. *Proc. ACM Program. Lang.* 3, POPL, Article 64 (Jan. 2019), 30 pages. https://doi.org/10.1145/3290377

[36] The Rocq Development Team. 2024. *The Rocq Prover*. https://doi.org/10.5281/zenodo.11551307

[37] Noam Zilberstein, Dexter Kozen, Alexandra Silva, and Joseph Tassarotti. 2025. A Demonic Outcome Logic for Randomized Nondeterminism. *Proc. ACM Program. Lang.* 9, POPL, Article 19 (Jan. 2025), 30 pages. https://doi.org/10.1145/3704855

[38] Noam Zilberstein, Alexandra Silva, and Joseph Tassarotti. 2024. Probabilistic Concurrent Reasoning in Outcome Logic: Independence, Conditioning, and Invariants. arXiv:2411.11662 [cs.LO] https://arxiv.org/abs/2411.11662

A Modular Proof of *conTwoAdd*

In this section, we show in more detail how to prove *conTwoAdd* with the HOCAP-style specifications of the randomized counter module (see Figure 5).

Before we proceed, we present a selection of side conditions of the abstract predicates in Figure 12 which we previously omitted in Figure 5. The first side condition expresses that the counter representation predicate is persistent, which means that it is duplicable so that clients can share it among several threads. We then have a series of side conditions regarding the *cauth* and *cfrag* abstract predicates, which are used to keep track of the abstract state of the counter. The first condition states that *cfrag* abstract predicates can be combined by adding their arguments together. The next condition states that if we hold both the *cauth* and *cfrag* resource and the fraction of the *cfrag* is exactly 1, the value from both predicates agree. The last side condition describes how we can update the abstract state of a counter: if we have a *cauth* and a *cfrag* predicate with the same ghost name, we can update the predicates by incrementing the values of both by a constant x.

$$
\begin{aligned}
C\ \iota\ \gamma\ c &\twoheadrightarrow \Box\, C\ \iota\ \gamma\ c \\
cfrag\ \gamma\ f\ z &\twoheadrightarrow cfrag\ \gamma\ f'\ z' \twoheadrightarrow cfrag\ \gamma\ (f+f')\ (z+z') \\
cauth\ \gamma\ z &\twoheadrightarrow cfrag\ \gamma\ 1\ z' \twoheadrightarrow z' = z \\
cauth\ \gamma\ z &\twoheadrightarrow cfrag\ \gamma\ f\ z' \twoheadrightarrow \Rrightarrow cauth\ \gamma\ (z+x) \twoheadrightarrow cfrag\ \gamma\ f\ (z'+x)
\end{aligned}
$$

Fig. 12. Selection of Side Conditions on Abstract Predicates

Recall that since the new specification of the randomized concurrent counter utilizes tapes, the *conTwoAdd* client is annotated to use the abstract tapes.

$$
\begin{aligned}
conTwoAdd \triangleq\ & \mathsf{let}\ c = createCntr() \ \mathsf{in} \\
& \Big(\ \mathsf{let}\ \kappa = createCtape() \ \mathsf{in}\ incrCntr\ c\ \kappa \ \Big|\Big|\ \mathsf{let}\ \kappa = createCtape() \ \mathsf{in}\ incrCntr\ c\ \kappa\ \Big); \\
& readCntr\ c
\end{aligned}
$$

We now prove that the return value is 0 with a probability of 1/16, with the Coneris Hoare triple: $\{\lightning(1/16)\}\ conTwoAdd\ \{v.v > 0\}$.

We first consider the invariant used to track the change in shared state during the parallel composition. We use two states S_0 and $S_1(n)$ (of some inductive type T) to track the state of the threads, with S_0 representing the state where the thread has not sampled a value yet and $S_1(n)$ representing it sampled n. Note that we do not need an additional state to track whether the sampled value has been added into the counter, because that can be tracked by the resource *cfrag*. We use the invariant I shown below to capture the shared state of the two threads. Notice that the invariant I makes use of the *exclusive-authoritative* ghost resource algebra, which consists of the *authoritative* part $\bullet x$ and the *fragment* part $\circ x$. We omit the definition and properties of this resource and we refer readers to Jung et al. [25] for more information.

$$
\begin{aligned}
\mathsf{sampled}\ s &\triangleq \mathsf{match}\ s\ \mathsf{with}\ \ S_0 \Rightarrow \mathsf{None}\ |\ S_1(n) \Rightarrow \mathsf{Some}\ n\ \mathsf{end} \\
\mathsf{onePositive}\ s_1\ s_2 &\triangleq \exists n.\ n > 0 \wedge (\mathsf{sampled}\ s_1 = \mathsf{Some}\ n \vee \mathsf{sampled}\ s_2 = \mathsf{Some}\ n) \\
I(\gamma_1, \gamma_2) &\triangleq \exists (s_1\ s_2 : T).\ \boxed{\bullet s_1}^{\gamma_1} * \boxed{\bullet s_2}^{\gamma_2} * \\
&\qquad \mathsf{if}\ \mathsf{onePositive}\ s_1\ s_2\ \mathsf{then}\ \lightning(0) \\
&\qquad \mathsf{else}\ \lightning\left(4^{(\mathsf{bool\_to\_nat}(\mathsf{sampled}\ s_1 = \mathsf{Some}\,0) + \mathsf{bool\_to\_nat}(\mathsf{sampled}\ n_2 = \mathsf{Some}\,0) - 2)}\right)
\end{aligned}
$$

We now show how to prove the specification of *conTwoAdd* using the invariant previously defined. After stepping through the code up until the parallel composition component, and allocating the necessary resources and invariant, we arrive at the following proof obligation:

$$\left\{ \begin{array}{l} C\ \iota\ \gamma\ c * \mathit{cfrag}\ \gamma\ 1\ 0 * \boxed{I(\gamma_1,\gamma_2)}^{\iota'} * \\ \qquad \boxed{\circ S_0}^{\gamma_1} * \boxed{\circ S_0}^{\gamma_2} \end{array} \right\}\ \begin{array}{l} \mathsf{let}\ldots \mid\mid\mid \mathsf{let}\ldots; \\ \mathit{readCntr}\ c \end{array}\ \{v.\, v > 0\}$$

We can apply the side condition of *cfrag* to split it between the two threads and apply the rule for parallel composition which leaves us with the following three obligations:

$$\left\{C\ \iota\ \gamma\ c * \boxed{I(\gamma_1,\gamma_2)}^{\iota'} * \mathit{cfrag}\ \gamma\ 0.5\ 0 * \boxed{\circ S_0}^{\gamma_1}\right\}\ \mathsf{let}\ldots\ \left\{\exists n.\ \mathit{cfrag}\ \gamma\ 0.5\ n * \boxed{\circ S_1(n)}^{\gamma_1}\right\} \tag{10}$$

$$\left\{C\ \iota\ \gamma\ c * \boxed{I(\gamma_1,\gamma_2)}^{\iota'} * \mathit{cfrag}\ \gamma\ 0.5\ 0 * \boxed{\circ S_0}^{\gamma_2}\right\}\ \mathsf{let}\ldots\ \left\{\exists n.\ \mathit{cfrag}\ \gamma\ 0.5\ n * \boxed{\circ S_2(n)}^{\gamma_1}\right\} \tag{11}$$

$$\left\{ \begin{array}{l} C\ \iota\ \gamma\ c * \mathit{cfrag}\ \gamma\ 1\ (n_1+n_2) * \boxed{I(\gamma_1,\gamma_2)}^{\iota'} * \\ \qquad \boxed{\circ S_1(n_1)}^{\gamma_1} * \boxed{\circ S_1(n_2)}^{\gamma_2} \end{array} \right\}\ \mathit{readCntr}\ c\ \{v.\, 0 < v\} \tag{12}$$

Let us first focus on Equation (10). We first apply the specification for *createCtape* to create an empty tape resource and we arrive at the following obligation.

$$\left\{ \begin{array}{l} C\ \iota\ \gamma\ c * \boxed{I(\gamma_1,\gamma_2)}^{\iota'} * \mathit{cfrag}\ \gamma\ 0.5\ 0 * \\ \qquad \boxed{\circ S_0}^{\gamma_1} * \mathit{ctape}\ \kappa\ \epsilon \end{array} \right\}\ \mathit{incrCntr}\ c\ \kappa\ \left\{\exists n.\ \mathit{cfrag}\ \gamma\ 0.5\ n * \boxed{\circ S_1(n)}^{\gamma_1}\right\}$$

Now that we have a *ctape* predicate on our hands, we can presample a value onto it so that it can be used for the *incrCntr* method later. Specifically, we perform the following probabilistic update:

$$\boxed{I(\gamma_1,\gamma_2)}^{\iota'} * \boxed{\circ S_0}^{\gamma_1} * \mathit{ctape}\ \kappa\ \epsilon \twoheadrightarrow \rightsquigarrow_{\top} \exists n. \boxed{\circ S_1(n)}^{\gamma_1} * \mathit{ctape}\ \kappa\ [n]$$

This probabilistic update proposition is proven by first applying the probabilistic update modality version of INV-OPEN where we access the resources within the invariant, and subsequently updating the authoritative resource pairs from the state S_0 to $S_1(n)$ to track the value n presampled onto the tape. After this probabilistic update, we are left with the following obligation:

$$\left\{ \begin{array}{l} C\ \iota\ \gamma\ c * \boxed{I(\gamma_1,\gamma_2)}^{\iota'} * \mathit{cfrag}\ \gamma\ 0.5\ 0 * \\ \qquad \boxed{\circ S_1(n)}^{\gamma_1} * \mathit{ctape}\ \kappa\ [n] \end{array} \right\}\ \mathit{incrCntr}\ c\ \kappa\ \left\{\exists n.\ \mathit{cfrag}\ \gamma\ 0.5\ n * \boxed{\circ S_1(n)}^{\gamma_1}\right\}$$

The rest of the proof then follows almost directly by applying the new specification for *incrCntr* and choosing $Q\ z \triangleq \mathit{cfrag}\ \gamma\ 0.5\ n$. The second obligation (Equation (11)), representing the behavior of the second thread, is proven in an almost identical fashion.

Let us now focus on the last obligation (Equation (12)). To prove that the return value is positive, we apply the specification of *readCntr*, choosing $Q\ v \triangleq v > 0$, leaving us with the following view shift obligation for the precondition:

$$\begin{array}{l} \boxed{I(\gamma_1,\gamma_2)}^{\iota'} * \mathit{cfrag}\ \gamma\ 1\ (n_1+n_2) * \boxed{\circ S_1(n_1)}^{\gamma_1} * \boxed{\circ S_1(n_2)}^{\gamma_2} \twoheadrightarrow \\ (\forall z.\ \mathit{cauth}\ \gamma\ z \twoheadrightarrow \Rightarrow_{\top\setminus\iota} \mathit{cauth}\ \gamma\ z * z > 0) \end{array}$$

We do a case split on the values of n_1 and n_2. If they are both 0, we can open the invariant $I(\gamma_1,\gamma_2)$ to access a $\lightning(1)$ error credit to derive a contradiction with ERR-1. Otherwise, using the rules for *cauth* and *cfrag*, we can show that the values in the *cauth* and *cfrag* predicates coincide, i.e. they are both $n_1 + n_2$ and must be positive, which completes the proof.

B HOCAP-style Specification with Error Redistribution

In §5.1, we presented a HOCAP-style specification that does not expose presampling tapes as an abstract predicate (see Figure 4). Although in §5.2 we showed that it is less general than the specification with *ctape* shown in Figure 5, in this section, we briefly explain how to use the specification to prove clients of the module and how to show various implementations meet the specification.

B.1 Implementation

We first show three possible implementations of the module that mirror those shown in §5.3. For I_1, we do not need to allocate any tapes and we sample from rand 3 directly. However, note

$$
\begin{aligned}
\mathit{incrCntr}_1 &\triangleq \lambda\, l.\ \mathsf{faa}\, l\ (\mathsf{rand}\, 3)\\
\mathit{incrCntr}_2 &\triangleq \lambda\, l.\ \mathsf{let}\ \kappa = \mathsf{tape}\, 1\ \mathsf{in}\\
&\qquad\qquad \mathsf{faa}\, l\ (\mathsf{rand}\, \kappa\ 1 * 2 + \mathsf{rand}\, \kappa\ 1)\\
\mathit{incrCntr}_3 &\triangleq \lambda\, l.\ \mathsf{let}\ \kappa = \mathsf{tape}\, 4\ \mathsf{in}\\
&\qquad\qquad (\mathsf{rec}\ f\ \kappa =\\
&\qquad\qquad \mathsf{let}\ x = \mathsf{rand}\, \kappa\ 4\ \mathsf{in}\\
&\qquad\qquad \mathsf{if}\ x < 4\ \mathsf{then}\ \mathsf{faa}\, l\ x\ \mathsf{else}\ f\ \kappa)\ \kappa
\end{aligned}
$$

Fig. 13. Three Implementations of Increment

that for I_2 and I_3, we have to create a tape internally and sample from it. This is because the randomization within the two implementations occurs over various steps even if it acts "logically atomic". By adding extra ghost code that utilizes tapes, we are able to reason about the randomness asynchronously, which we demonstrate in later subsections.

B.2 Verification of Client of HOCAP-style Specification with Error Redistribution

We now verify the *conTwoAdd* example in §5.2 with the specification from Figure 4. Since tapes are not exposed in this specification, *conTwoAdd* is written without explicit allocation of *ctape*.

$$
\begin{aligned}
\mathit{conTwoAdd} \triangleq\ & \mathsf{let}\ c = \mathit{createCntr}()\ \mathsf{in}\\
& (\mathit{incrCntr}\ c \mid\mid\mid \mathit{incrCntr}\ c);\\
& \mathit{readCntr}\ c
\end{aligned}
$$

As before, we want to verify that the final read value is positive, with error probability 1/16, which we show with the following Coneris Hoare triple.

$$\{\lightning\,(1/16)\}\ \mathit{conTwoAdd}\ \{v.v > 0\}$$

In fact the proof works almost identically to that presented in §5.2. For example, the states and invariants used to track the shared state of the two parallel threads are identical to the ones used before.

We begin by stepping through the code up until the parallel composition component, and after allocating the necessary resources and invariant, we arrive at the following proof obligation:

$$\Big\{C\ \iota\ \gamma\ c * \mathit{cfrag}\ \gamma\ 1\ 0 * \boxed{I(\gamma_1,\gamma_2)}^{\iota'} * \boxed{\circ S_0}^{\gamma_1} * \boxed{\circ S_0}^{\gamma_2}\Big\}$$
$$\mathit{incrCntr}\ c \,|||\, \mathit{incrCntr}\ c;$$
$$\mathit{readCntr}\ c$$
$$\{v.\, v > 0\}$$

We can apply the side condition of *cfrag* to split it between the two threads and apply the rule for parallel composition, leaving us with the following three obligations:

$$\Big\{C\ \iota\ \gamma\ c * \boxed{I(\gamma_1,\gamma_2)}^{\iota'} * \mathit{cfrag}\ \gamma\ 0.5\ 0 * \boxed{\circ S_0}^{\gamma_1}\Big\}\ \mathit{incrCntr}\ c\ \{\exists n.\ \mathit{cfrag}\ \gamma\ 0.5\ n * \boxed{\circ S_1(n)}^{\gamma_1}\} \tag{13}$$

$$\Big\{C\ \iota\ \gamma\ c * \boxed{I(\gamma_1,\gamma_2)}^{\iota'} * \mathit{cfrag}\ \gamma\ 0.5\ 0 * \boxed{\circ S_0}^{\gamma_2}\Big\}\ \mathit{incrCntr}\ c\ \{\exists n.\ \mathit{cfrag}\ \gamma\ 0.5\ n * \boxed{\circ S_2(n)}^{\gamma_1}\} \tag{14}$$

$$\Big\{C\ \iota\ \gamma\ c * \mathit{cfrag}\ \gamma\ 1\ (n_1+n_2) * \boxed{I(\gamma_1,\gamma_2)}^{\iota'} * \boxed{\circ S_1(n_1)}^{\gamma_1} * \boxed{\circ S_1(n_2)}^{\gamma_2}\Big\}\ \mathit{readCntr}\ c\ \{v.\, 0 < v\} \tag{15}$$

We focus only on the first obligation; the second obligation follows similarly and the third obligation is similar to the proof of Equation (12) in §5.2. From Equation (13), we apply the specification of *incrCntr* directly, choosing $Q\ \varepsilon\ \mathcal{F}\ n\ z \triangleq \mathit{cfrag}\ \gamma\ 0.5\ n * \boxed{\circ S_1(n)}^{\gamma_1}$. It then suffices to prove the following view shift for the precondition of the specification:

$$\boxed{I(\gamma_1,\gamma_2)}^{\iota'} * \mathit{cfrag}\ \gamma\ 0.5\ 0 * \boxed{\circ S_0}^{\gamma_1} \mathrel{-\!\!*}$$
$${}_{\mathcal{E}}\!\Rrightarrow_{\emptyset} \exists \varepsilon\ \mathcal{F}.\ \lightning(\varepsilon) * (\mathbb{E}_{\mathfrak{U}3}[\mathcal{F}] \le \varepsilon) *$$
$$(\forall x.\, 0 \le x < 4 \mathrel{-\!\!*} \lightning(\mathcal{F}(x)) \mathrel{-\!\!*} {}_{\emptyset}\!\Rrightarrow_{\mathcal{E}}$$
$$(\forall z.\ \mathit{cauth}\ \gamma\ z \mathrel{-\!\!*} \Rrightarrow_{\mathcal{E}\setminus\iota} \mathit{cauth}\ \gamma\ (z+x) * \mathit{cfrag}\ \gamma\ 0.5\ x * \boxed{\circ S_1(x)}^{\gamma_1}))$$

We first open our invariant $I(\gamma_1,\gamma_2)$ while stripping away the ${}_{\mathcal{E}}\!\Rrightarrow_{\emptyset}$ mask, which allows us to access the error credit stored in the invariant. After choosing the right $\mathcal{F}$ based on a case analysis on the state of the right thread (which we omit for brevity), we update the authoritative resource pairs from $\boxed{\bullet S_0}^{\gamma_1} * \boxed{\circ S_0}^{\gamma_1}$ to $\boxed{\bullet S_1(x)}^{\gamma_1} * \boxed{\circ S_1(x)}^{\gamma_1}$ and re-establish the invariant I while removing the ${}_{\emptyset}\!\Rrightarrow_{\mathcal{E}}$ mask, leaving us with the following state:

$$\boxed{I(\gamma_1,\gamma_2)}^{\iota'} * \mathit{cfrag}\ \gamma\ 0.5\ 0 * \boxed{\circ S_1(x)}^{\gamma_1} \mathrel{-\!\!*}$$
$$\mathit{cauth}\ \gamma\ z \mathrel{-\!\!*} \Rrightarrow_{\mathcal{E}\setminus\iota} \mathit{cauth}\ \gamma\ (z+x) * \mathit{cfrag}\ \gamma\ 0.5\ x * \boxed{\circ S_1(x)}^{\gamma_1}$$

After incrementing both the *cauth* and *cfrag* components by exactly x through the $\Rrightarrow_{\mathcal{E}\setminus\iota}$ mask (which is possible by the side conditions of *cauth* and *cfrag*), we can then directly establish the final goal.

B.3 Proving that I_1, I_2, and I_3 Satisfy the HOCAP-style Specification with Error Redistribution

We now briefly describe how each of the three randomized counter implementations meets the specification with error redistribution in Figure 4. The concrete definitions for the abstract predicates are actually identical to those used in the proof of §5.2. For example, the counter predicate is still defined as

$$\boxed{\exists (l : \mathit{Loc})(n : \mathrm{nat}).\ c = l * l \mapsto n * \mathit{cauth}\ \gamma\ n}^{\iota}$$

It then suffices to show that the functions *createCntr*, *incrCntr*, and *readCntr* satisfy the HOCAP-style specification. We focus on the *incrCntr* specification since it is the most complicated; the other two functions can be verified in a similar, if not easier, fashion.

For I_1, after symbolically stepping through the program, we are left with the following obligation:

$$\left\{ \begin{array}{l} C\ \iota\ \gamma\ c * ({}_{\mathcal{E}}\Rrightarrow_{\emptyset} \exists \varepsilon\ \mathcal{F}.\ \lightning(\varepsilon)\ * (\mathbb{E}_{\mathfrak{U}3}[\mathcal{F}] \le \varepsilon) * \\ (\forall x.\ 0 \le x < 4 \mathrel{-\!\!*} \lightning(\mathcal{F}(x)) \mathrel{-\!\!*} {}_{\emptyset}\Rrightarrow_{\mathcal{E}} \\ (\forall z.\ cauth\ \gamma\ z \mathrel{-\!\!*} \Rrightarrow_{\mathcal{E}\setminus\iota} cauth\ \gamma\ (z + x) * Q\ \varepsilon\ \mathcal{F}\ x\ z))) \end{array} \right\}$$

$$\mathsf{faa}\ c\ (\mathsf{rand}\ 3)\ \{z. \exists \varepsilon\ \mathcal{F}\ x.\ Q\ \varepsilon\ \mathcal{F}\ x\ z\}_{\mathcal{E}}$$

We first open the ${}_{\mathcal{E}}\Rrightarrow_{\emptyset}$ mask around the atomic rand 3 operation. After opening the first view shift, we are given some $\lightning(\varepsilon)$ and some $\mathcal{F}$ such that the expected sum of $\mathcal{F}$ is smaller than ε. We then apply HT-RAND-EXP to distribute the errors across the various results and close the ${}_{\emptyset}\Rrightarrow_{\mathcal{E}}$ mask, leaving us with the following obligation:

$$\left\{\ C\ \iota\ \gamma\ c * (\forall z.\ cauth\ \gamma\ z \mathrel{-\!\!*} \Rrightarrow_{\mathcal{E}\setminus\iota} cauth\ \gamma\ (z + x) * Q\ \varepsilon\ \mathcal{F}\ x\ z)\ \right\}$$

$$\mathsf{faa}\ c\ x\ \{z. \exists \varepsilon\ \mathcal{F}\ x.\ Q\ \varepsilon\ \mathcal{F}\ x\ z\}_{\mathcal{E}}$$

Since the faa operation is atomic, we can open the invariant C around the expression, resulting in this goal:

$$\left\{\ l \mapsto n * cauth\ \gamma\ n * (\forall z.\ \ldots)\ \right\}$$

$$\mathsf{faa}\ l\ x\ \{z. \exists (n : \mathsf{nat}).\ l \mapsto n * cauth\ \gamma\ n * \exists \varepsilon\ \mathcal{F}\ x.\ Q\ \varepsilon\ \mathcal{F}\ x\ z\}_{\mathcal{E}\setminus\iota}$$

The rest of the proof follows nicely from the proof rule for the faa operation, completing the proof.

For I_2, we similarly step through the program, where we additionally allocate a tape κ, and thus we arrive at the following goal:

$$\left\{ \begin{array}{l} C\ \iota\ \gamma\ c * \kappa \hookrightarrow (1, \epsilon) * ({}_{\mathcal{E}}\Rrightarrow_{\emptyset} \exists \varepsilon\ \mathcal{F}. \\ \lightning(\varepsilon)\ * (\mathbb{E}_{\mathfrak{U}3}[\mathcal{F}] \le \varepsilon) * \\ (\forall x.\ 0 \le x < 4 \mathrel{-\!\!*} \lightning(\mathcal{F}(x)) \mathrel{-\!\!*} {}_{\emptyset}\Rrightarrow_{\mathcal{E}} \\ (\forall z.\ cauth\ \gamma\ z \mathrel{-\!\!*} \Rrightarrow_{\mathcal{E}\setminus\iota} cauth\ \gamma\ (z + x) * Q\ \varepsilon\ \mathcal{F}\ x\ z))) \end{array} \right\}$$

$$\mathsf{faa}\ l\ (\mathsf{rand}\ \kappa\ 1 * 2 + \mathsf{rand}\ \kappa\ 1)\ \{z. \exists \varepsilon\ \mathcal{F}\ x.\ Q\ \varepsilon\ \mathcal{F}\ x\ z\}_{\mathcal{E}}$$

From here, we directly open the ${}_{\mathcal{E}}\Rrightarrow_{\emptyset}$ and access the $\lightning(\varepsilon)$ error credit. Then, unlike what we did for I_1, here we perform a *probabilistic update* where we presample two values v_1, v_2 onto the tape κ, and we distribute $\lightning(v_1 * 2 + v_2)$ for each branch, i.e., we update the resources via the following lemma (in this instance, $\vec{n}$ is instantiated to be the empty tape list ϵ). This follows from Equation (7) which we proved previously.

After closing the ${}_{\emptyset}\Rrightarrow_{\mathcal{E}}$ mask, we are left with the following obligation:

$$\left\{ \begin{array}{l} C\ \iota\ \gamma\ c * \kappa \hookrightarrow (1, [v_1, v_2]) * \\ (\forall z.\ cauth\ \gamma\ z \mathrel{-\!\!*} \Rrightarrow_{\mathcal{E}\setminus\iota} cauth\ \gamma\ (z + v_1 * 2 + v_2) * Q\ \varepsilon\ \mathcal{F}\ (v_1 * 2 + v_2)\ z) \end{array} \right\}$$

$$\mathsf{faa}\ l\ (\mathsf{rand}\ \kappa\ 1 * 2 + \mathsf{rand}\ \kappa\ 1)\ \{z. \exists \varepsilon\ \mathcal{F}\ x.\ Q\ \varepsilon\ \mathcal{F}\ x\ z\}_{\mathcal{E}}$$

We can then read the values of the tape directly for both samples:

$$\left\{ \begin{array}{l} C\ \iota\ \gamma\ c * \kappa \hookrightarrow (1, \epsilon) * \\ (\forall z.\ cauth\ \gamma\ z \mathrel{-\!\!*} \Rrightarrow_{\mathcal{E}\setminus\iota} cauth\ \gamma\ (z + v_1 * 2 + v_2) * Q\ \varepsilon\ \mathcal{F}\ (v_1 * 2 + v_2)\ z) \end{array} \right\}$$

$$\mathsf{faa}\ l\ (v_1 * 2 + v_2)\ \{z. \exists \varepsilon\ \mathcal{F}\ x.\ Q\ \varepsilon\ \mathcal{F}\ x\ z\}_{\mathcal{E}}$$

From here, the fetch-and-atomic-add step is similar to that for the I_1 implementation.

$$\begin{pmatrix} \forall \varepsilon\ \mathcal{F}\ \iota\ \mathcal{E}\ \kappa\ \gamma\ \vec{n}. \\ (\mathbb{E}_{\mathcal{U}_N}[\mathcal{F}] \leq \varepsilon) \twoheadrightarrow \\ \iota \in \mathcal{E} \twoheadrightarrow \\ \mathit{isRand}\ \iota\ \gamma \twoheadrightarrow \\ \mathit{randTape}\ \kappa\ \vec{n}\ \gamma \twoheadrightarrow \\ \lightning(\varepsilon) \twoheadrightarrow \\ \rightsquigarrow_{\mathcal{E}} \exists n.\ \lightning(\mathcal{F}(n)) * \mathit{randTape}\ \kappa\ (\vec{n} \cdot [n])\ \gamma \end{pmatrix}$$

(a) Presampling specification

$$\begin{array}{l} \forall \iota\ \gamma\ \kappa\ n\ \vec{n}\ \mathcal{E}. \\ \{\ \iota \in \mathcal{E} * \mathit{isRand}\ \iota\ \gamma * \mathit{ctape}\ \kappa\ (n \cdot \vec{n})\ \} \\ \mathit{randf}\ \kappa\ \{z.z = n * \mathit{randTape}\ \kappa\ \vec{n}\}_{\mathcal{E}} \end{array}$$

(b) *randf* specification

Fig. 14. Selection of Specification of Rand Module

The proof of I_3 is very similar to that of I_2, except for the presampling step after allocating the tape resource. In particular we want to show that we can repeatedly presample enough values into the tape such that the last element is smaller than 4 and all values beforehand are 4, while distributing the error credit according to the final value. Here we use Equation (8) proved previously to do so.

After performing the probabilistic update on the tape (such that it contains an "accepted" value at the end), we can then step through the rest of the program, looping repeatedly until we reach the final "accepted" value and establish the postcondition.

C Other Case Studies

C.1 Rand Module

For the random counter module introduced previously in §5.1, we identified three distinct implementations (§5.3) that sample randomness from a uniform distribution for the *incrCntr* operation, e.g. we can directly call a single rand (I_1), chain various rands together (I_2), or use a rejection sampler method where we repeatedly sample until we obtain a desirable value (I_3). We now define a general interface, which we refer to as the *Rand module*, that captures what it means to sample from a uniform distribution atomically, and show that several implementations satisfy it. In later case studies, we use this interface to verify larger programs, to highlight the usability of this module and to demonstrate modular reasoning.

The Rand module is parameterized by a natural number N, which is the range of the uniform distribution (we are sampling uniformly from $\{0, \ldots, N\}$). The interface exposes two functions, *randAllocate* and *randf*, which creates a tape and samples from it, respectively. It also describes various abstract predicates, their side conditions, and specifications of the functions, most notably the specification that allows clients to presample into the abstract tape *randTape*, and reading from it with *randf*, which we present in Figure 14.

The first condition states that when given the *isRand* invariant, a *randTape* abstract predicate, and some error credits, we can append a value at the end of the tape and split the errors in an expectation-preserving way, similar to the presampling specification presented in the random counter module. The second condition states that given the *isRand* invariant and a non-empty tape, we can run *randf* on the tape to deterministically pop the tape and return its first element.

By choosing concrete definitions for the abstract predicates of this module, one can show that various implementations of random samplers satisfy this Rand module specification; e.g. we proved that a rejection sampler meets the specification of the Rand module (the proof is similar to showing that implementation I_3 of the randomized counter module meets its specification).

C.2 Concurrent Amortized Collision-free Hash

When proving the correctness of randomized data structures, it is useful to assume that a hash function is collision-free, in that different input keys for the function return different hash values. In reality, collisions might occur but with very low probabilities.

Here, we first consider a concurrent collision-free hash, one that can be shared among many threads, and each thread pays error credits to avoid collisions for every presampling action. Because we want to be able to use the hash in a concurrent context, the specification of the hash is written in a HOCAP-style with presampling tapes exposed to allow modular reasoning. We implement a concurrent model of the idealized hash function under the uniform hash assumption [11]. The assumption states that the hash function *hashf* mapping sets of keys K to hash values V is a random oracle, in that for each key $k \in K$, the hash value $h(K)$ is sampled uniformly from V independently of all other keys. We implement this model as a tuple containing a lock and a mutable map lm, choosing K and V to be $\{0, \ldots, N\}$. The main hash function *hashf* is shown below. The lock is acquired and released around the body of the hash function to ensure that at most one thread is changing the state of the mutable map. In the critical section, if the key k has been hashed before, we directly return $lm(k)$. Otherwise, we sample a fresh value uniformly from V with the *randf* function defined in Appendix C.1, read a value from the tape κ, store it in $lm(k)$, and return it at the end.

```
hashf (lo, lm) k κ ≜ acquire lo;
                     let v = match get lm k with
                     | Some(b) ⇒ b
                     | None ⇒  let b = randf κ in
                               set lm k b;
                               b
                     end in
                     release lo; v
```

We show the presampling specification and the specification for *hashf* in Figure 15. To achieve collision-freedom, we need to ensure that every value we presample to a tape generated by the hash is different from any value previously presampled to *all* tapes generated by the hash. To be precise, suppose we have presampled a total of s values on all tapes of the hash. If we want to presample a new value to a tape, we need to pay at least $\lightning\left(\frac{s}{N+1}\right)$ to sample a unique value different from all values presampled before. To keep track of all the values sampled before, the interface introduces a *hashsize* abstract predicate that stores the set of all values that has been presampled before. To presample onto a tape for the collision-free hash, we need to additionally pass in a *hashsize* predicate to determine the amount of error needed to avoid the previous presampled-values. The *hashf* specification is defined in almost the same way as the *lazyRandf* specification, the view shift in the precondition performs a case split to determine whether a key has been hashed before by looking into the mutable map.

We also used this specification to derive an *amortized* version of the collision-free hash, which we show in Figure 16. This hash specification has two main advantages. Firstly, clients do not need to pass a *hashsize* predicate as a precondition for presampling into the tape. In addition, the error credit $\varepsilon_A(N, M)$ to be paid is constant as it is amortized across a fixed number of insertions M that is decided in advance. To keep track of the maximum number of times the hash is used, clients need to give up a single *hashToken* predicate; exactly M number of these *hashTokens* are generated when the hash is initialized. The proof of this more complex specification is similar to that in Aguirre

$$\left(\begin{array}{l}\forall \varepsilon_O\ P\ \iota\ \mathcal{E}\ \kappa\ \gamma\ \vec{n}\ s\ h.\\ ((s+(N+1-s)\varepsilon_O)/(N+1) \le \varepsilon) \twoheadrightarrow\\ \iota \in \mathcal{E} \twoheadrightarrow\\ \mathit{isHash}\ h\ P\ \iota\ \gamma \twoheadrightarrow\\ \mathit{hashTape}\ \kappa\ \vec{n}\ \gamma \twoheadrightarrow\\ \mathit{hashsize}\ s\ \gamma \twoheadrightarrow\\ \lightning(\varepsilon) \twoheadrightarrow\\ \Rightarrow\!\!\!\!\!\sim_{\mathcal{E}} \exists n.\ \lightning(\varepsilon_O) * \mathit{hashsize}\ (s+1)\ \gamma *\\ \mathit{hashTape}\ \kappa\ (\vec{n} \cdot [n])\ \gamma\end{array}\right)$$

(a) Presampling Specification

$$\begin{array}{l}\forall \iota\ \gamma\ h\ P\ \kappa\ k\ Q_1\ Q_2.\\ \left\{\begin{array}{l}\mathit{isHash}\ h\ P\ \iota\ \gamma *\\ (\forall m.\ P\ m \twoheadrightarrow \mathit{hashAuth}\ m\ \gamma \twoheadrightarrow\\ \Rightarrow\!\!\!\!\!\sim_{\top}\ \mathsf{match}\ m!!k\ \mathsf{with}\\ \mid \mathsf{Some}\ v \Rightarrow P\ m * \mathit{hashAuth}\ n\ \gamma * Q_1\ n\\ \mid \mathsf{None} \Rightarrow \exists n\ \vec{n}.\ \mathit{hashTape}\ \kappa\ (n \cdot \vec{n})\ \gamma *\\ (\mathit{hashTape}\ \kappa\ \vec{n}\ \gamma \twoheadrightarrow\\ \Rightarrow\!\!\!\!\!\models_{\top} P\ (m[k := n]) *\\ \mathit{hashAuth}\ (m[k := n])\ \gamma * Q_2\ n\ \vec{n})\\ \mathsf{end})\end{array}\right\}\\ \mathit{hashf}\ h\ k\ \kappa\ \{x.Q_1\ x \vee \exists \vec{n}.\ Q_2\ x\ \vec{n}\}_{\mathcal{E}}\end{array}$$

(b) *hashf* Specification

Fig. 15. Selection of Specification of the Collision-free Hash

et al. [3] which we omit here. We emphasize that this amortized specification can be *derived* from the non-amortized specification (Figure 15) without taking into account how the concurrent hash is implemented.

$$\left(\begin{array}{l}\forall \varepsilon_O\ P\ \iota\ \mathcal{E}\ \kappa\ \gamma\ \vec{n}\ h.\\ \iota \in \mathcal{E} \twoheadrightarrow\\ \mathit{isHash}\ h\ P\ \iota\ \gamma \twoheadrightarrow\\ \mathit{hashTape}\ \kappa\ \vec{n}\ \gamma \twoheadrightarrow\\ \mathit{hashToken}\ 1\ \gamma \twoheadrightarrow\\ \lightning(\varepsilon_A(N,M)) \twoheadrightarrow\\ \Rightarrow\!\!\!\!\!\sim_{\mathcal{E}} \exists n.\ \mathit{hashTape}\ \kappa\ (\vec{n} \cdot [n])\ \gamma\end{array}\right)$$

Fig. 16. Amortized Presampling Specification

Just like the specification presented in §6.3, the specification for both the non-amortized and amortized concurrent collision-free hashes uses the probabilistic update modality in the view shift in its *hashf* specification to allow presampling to occur within the *hashf* body. As an example, consider the following program (similar to *lazyRace* in §6.3) and its specification.

$$\begin{array}{l}\{\lightning(\varepsilon_A(N,M))\}\\ \quad \mathsf{let}\ h = \mathit{initHash}\ ()\ \mathsf{in}\\ \quad (\mathit{hashf}\ h\ 0\ (\mathit{hashAllocTape}\ ()))\ |||\ (\mathit{hashf}\ h\ 0\ (\mathit{hashAllocTape}\ ()))\\ \{v.\exists n.\ v = (n,n)\}\end{array}$$

Here we create an amortized hash and spawn two threads that each creates a tape and uses the tape to hash the value 0. Because both threads are hashing the same key 0, it should be the case that we only need to pay one constant $\varepsilon_A(N,M)$ for the first hash operation. However we do not know which thread is scheduled first in advance, so we cannot perform the presampling in advance before the *hashf* call. The probabilistic update modality allows us to perform the presampling within the *hashf* call in the case where the value of $m!!0$ is None, indicating that this thread has been scheduled first to do the randomized sampling.