# Formal Certification of Randomized Algorithms


## Abstract

Randomized algorithms have broad applications throughout computer science. They also pose a challenge for formal verification: even intuitive properties of simple programs can have elaborate proofs, mixing program verification with probabilistic reasoning.

We present Ellora, a tool-assisted framework for the interactive verification of general properties of randomized algorithms. The central component of Ellora is a new and expressive program logic for imperative programs with adversarial code. In particular, the logic supports fine-grained reasoning about probabilistic loops by offering different reasoning principles according to the termination behavior, and assertions can model key notions from probability theory, such as probabilistic independence and expected values. We show soundness for the logic and develop an implementation on top of the EasyCrypt proof assistant.

We demonstrate the strength of Ellora in two ways. First, we embed several specialized logics into Ellora: an adaptation of greatest pre-expectation calculus from Kozen [32], Morgan et al. [39] (restricted to loop-free programs without non-determinism), the union bound logic from Barthe et al. [7], and a novel Hoare logic for reasoning about distribution laws and probabilistic independence. Second, we formally verify several classical randomized algorithms.


## 1. Introduction

Randomized algorithms are a fundamental objects of study in theoretical computer science, with broad applications to computational fields like cryptography and machine learning. They also present a challenging target for formal verification. Often presented as imperative programs, they satisfy appealing properties such as computational efficiency and various notions of probabilistic accuracy. While the properties often capture intuitive features, their correctness can be subtle—even simple properties may require intricate proofs using complex mathematical theorems.

While mathematical theorem can play a role in proving correctness for all programs, probabilistic or not, proofs for randomized algorithms frequently apply tools from a broad collection of concepts and results from probability theory, like distribution laws, probabilistic independence, and concentration bounds. As a consequence, a formal framework for reasoning about randomized algorithms should provide mechanisms for smoothly applying these common tools, as they are traditionally used in paper-and-pen proofs.

For deductive verification, where specifications are given by a pre-condition and a post-condition, a natural verification strategy is to let assertions be interpreted as sets of distributions over program states. This idea was first proposed by Ramshaw [42], and subsequently refined by other researchers [9, 15, 43]. However, existing systems have several shortcomings. First, typical examples of randomized algorithms and their properties are difficult to express. Existing program logics do not support assertions about general expected values, a fundamental part of many target properties, and restrict sampling to Boolean distributions. All other distributions, like the uniform distribution or the normal distribution, need to be simulated with loops.

Second, existing reasoning principles for proving specifications are also limited. Prior work does not consider reasoning about lossy programs, i.e. programs which terminate with probability strictly less than 1, and about programs with adversarial code, a natural concept in many applications from security and privacy (which naturally carry a notion of adversary), and from game theory and mechanism design (where adversaries model strategic agents). Furthermore, proofs commonly require low-level reasoning about the semantics of programs and assertions. For instance, reasoning about loops involves semantic side conditions that can be difficult to prove; many program logics use non-standard logical connectives to reason about random sampling and conditionals.

### The Ellora framework

In this paper we introduce Ellora, a mechanized framework for general-purpose, interactive reasoning about randomized algorithms. The central component is a new probabilistic program logic alleviating the shortcomings of previous work in several respects; we highlight the main novelties here.

*Reasoning about loops.* Proving a property of a probabilistic loop typically requires analyzing its termination behavior and establishing a loop invariant. Moreover, the class of loop invariants that can be soundly used depends on the termination behavior. We identify three classes of assertions that can be used for reasoning about probabilistic loops, and provide a proof rule for each one:

- arbitrary assertions for *certainly terminating* loops, i.e. loops that terminate in a finite amount of iterations;
- *topologically closed* assertions for *almost surely* terminating loops, i.e. loops that terminate with probability 1;
- *downwards closed* assertions for arbitrary loops.

Our definition of topologically closed assertion is reminiscent of Ramshaw [42]; the stronger notion of downwards closed assertion appears to be new.

Besides broadening the class of loops that can be analyzed, a diverse collection of rules can lead to simpler proofs. For instance, if the loop is certainly terminating, then there is no need to prove semantic side conditions. Likewise, there is no need to consider the termination behavior of the loop when the invariant is downwards and topologically closed. For example, in many applications, especially cryptography, the target property is that a "bad" event has low probability: $\Pr[E] \leq k$. This assertion is downwards and topologically closed, so it can be a sound loop invariant regardless of the termination behavior.

*Reasoning about adversaries.* *Adversaries* are special procedures, specified by an interface listing the concrete procedures that an adversary can call, along with restrictions like how many calls an adversary may make. Adversaries are widely used in cryptography, where security notions are described using experiments in which one or several adversaries interact with a challenger, and in game theory and mechanism design, where they are used for modeling strategic agents. Adversaries can also model *online* algorithms, where an external party interacts with an algorithm.

We provide several proof rules for reasoning about adversary calls. Similar to the case of **while** loops, different proof rules are used depending on the class of adversary considered. Our rules are significantly more general than previous considered rules for reasoning about adversaries. For instance, the rule for adversary used by Barthe et al. [7] is restricted to adversaries that cannot make oracle calls, and applies to a specialized program logic that only supports reasoning with a simple fact from probability theory called the union bound.

*Reasoning with specialized tools.* The first central goal of Ellora is providing support for specialized reasoning principles from

existing, "paper" proofs of randomized algorithms. These patterns do not always apply, but they are lightweight methods to prove specific types of commonly-used properties. By simplifying and organizing proofs about common properties, these patterns form an indispensable part of a toolkit for reasoning about randomized algorithms. We demonstrate support in Ellora for these principles by embedding several specialized logics that neatly capture specific aspects of probabilistic reasoning: the *union bound* logic by Barthe et al. [7] (for proving upper bounds on probabilities), the loop-free fragment of the *greatest pre-expectation calculus* due to McIver and Morgan [35] (for computing expectations), and an *independence and distribution law* logic. This last logic is new and potentially of independent interest, designed to reason about independence in a lightweight way that is common in paper proofs.

*Reasoning about probabilistic notions.* The second central goal of Ellora is general reasoning about common properties and notions from existing proofs, like probabilities, expected values, distribution laws and probabilistic independence. There is prior work covering some of these aspects, most notably for manipulating expected values, but it remains practically challenging to carry out general probabilistic arguments within a single system. We demonstrate Ellora on a collection of case studies, including textbook examples and a randomized routing algorithm. Our experience suggests that Ellora is capable of both expressing and reasoning about the toolbox of properties found in existing proofs, freely applying theorems from probability theory.

*Implementation.* We develop a full-featured implementation of our framework on top of EasyCrypt, a general-purpose proof assistant for reasoning about probabilistic programs. Assertions in our implementation have a concrete syntax, encoding a two-level assertion language. The first level contains state predicates—deterministic assertions about a single memory—while the second layer includes probabilistic assertions constructed from probabilities and expected values over discrete distributions. While the concrete language cannot express arbitrary predicates on distributions, it is a natural fit for all properties and invariants of randomized algorithms that we have encountered. More importantly, the assertion language supports several syntactic tools to simplify verification:

- an automated procedure for generating pre-conditions of non-looping commands, inspired from prior work on greatest pre-expectations [32, 39]; and
- syntactic conditions for the closedness and termination properties required for soundness of the loop rules, and syntactic conditions for soundness of the frame and adversary rules.

In order to carry out full verification of example algorithms in our implementation, we also develop a partial formalization of probability theory in Ellora, including common tools like concentration bounds (e.g., the Chernoff bound), Markov's inequality, and theorems about probabilistic independence.

### Contributions

To summarize, we present the following contributions.

- A probabilistic Hoare logic with general predicates on distributions, rules for handling different kinds of probabilistically terminating loops and procedure calls, and a mechanized proof of soundness for the logic;
- a concrete version of the logic with an assertion language suitable for syntactic tools, and an implementation within a general-purpose theorem prover;
- embeddings of three specialized reasoning tools: a core version of the greatest pre-expectation calculus from Morgan et al. [39], the union bound logic from Barthe et al. [7], and a novel Hoare logic for reasoning about distribution laws and probabilistic independence; and
- case studies demonstrating formal verification of randomized algorithms.

***Comparison with expectation-based techniques.*** To date, arguably the most mature systems for deductive verification of randomized algorithms are derived from *expectation-based* techniques. These systems consider *expectations*, functions $E$ from states to real numbers; the name comes by considering a program as mapping an input state $s$ to a distribution $\mu(s)$ on output states, when the expected value of $E$ on $\mu(s)$ is an expectation. Roughly speaking, expectation-based approaches offer deductive principles to compositionally apply the effect of the program to the expectation, transforming it until it can be analyzed purely mathematically. A probabilistic property, expressed as a formula involving probabilities and expectations, is proved by separately transforming each component in this way. Classical examples include PPDL [32] and pGCL [39]; the latter work also considers non-determinism. Expectation-based systems are very elegant and have a neat meta-theory. Moreover, they have been used for verifying several randomized algorithms. In particular, there have been efforts to mechanize these systems and to verify sophisticated case studies, e.g. Hurd [22], Hurd et al. [24].

A direct comparison with probabilistic Hoare logics is difficult, since the two approaches are quite different. In broad strokes, program logics can verify richer properties in one shot, have assertions that are easier to understand, and can make assertions about the input viewed as a distribution, while expectation-based approaches can transform expectations mechanically and reason about loops without semantic side conditions. By incorporating tools inspired by expectation-based techniques and designing loop rules with syntactic side conditions, we aim for the best of both worlds within Ellora.

## 2. Example: accuracy of private sum

```
proc psum (a[N]:int array):
  var s:int, l[N]:int array;
  s ← 0;
  for j ← 0 to N-1 do
    l[j] ←$ L_ε;
    s ← s + l[j] + a[j];
  return s
```

**Figure 1.** Private sum

We illustrate the style of reasoning that we want to capture in Ellora using a simple program inspired by differential privacy. The program (Fig. 1) computes the private sum of an array as follows: it draws samples from the Laplace distribution $\mathcal{L}_\epsilon$ for each element, and then adds each element with its noise to the current sum.

Our goal is to establish an accuracy bound for private sum: the return value $\mathtt{s}$, which holds the sum of $\mathtt{a}$, should be close to the true sum $\widehat{\mathtt{s}}$ of $\mathtt{a}$ with high probability. First, we establish the following loop invariant for each iteration $\mathtt{j}$:

$$\left(\mathtt{s} - \sum_{k=0}^{\mathtt{j}} \mathtt{a}[k] = \sum_{k=0}^{\mathtt{j}} \mathtt{l}[k]\right) \wedge \#(\mathtt{l}[0], \ldots, \mathtt{l}[\mathtt{j}]) \wedge \bigwedge_{1 \le i \le \mathtt{j}} \mathtt{l}[k] \sim \mathcal{L}_\epsilon$$

where the second conjunct states that $\mathtt{l}[0], \ldots, \mathtt{l}[\mathtt{j}]$ are independent random variables and the third conjunct states that they are all distributed according to $\mathcal{L}_\epsilon$. Next, we use the second and third conjuncts of the invariant to apply a *concentration bound*. This theorem gives a formula $T : (0, 1) \to \mathbb{R}$ such that for every failure probability $b \in (0, 1)$, $T(b)$ upper bounds the sum of $\mathtt{l}$ except with

probability $b$:

$$\Pr\left[|\mathsf{s} - \widehat{\mathsf{s}}| > T(b)\right] = \Pr\left[\left|\sum_{k=0}^{\mathsf{N-1}} \mathsf{l}[k]\right| > T(b)\right] \leq b$$

This simple example highlights two desirable features of a proof system for probabilistic programs:

1. the ability to both prove and use properties of distributions, like probabilistic independence and i.i.d. variables;
2. the ability to internalize basic theorems of probability, like concentration bounds.

To compare, using the concentration bound yields a better than the bound than simpler approaches like aHL [7], a lightweight program logic which can reason about accuracy of differentially private computations by using the *union bound* but cannot reason about independence. Expectation-based approaches (e.g., PPDL [32] or pGCL [39]) can in principle establish the same bound, but the verification strategy would be rather different from the proof we sketched above. Since there is no direct way to use concepts like probabilistic independence or concentration bounds, expectation-based proofs propagate probabilities like $\Pr\left[\left|\sum_{k=0}^{\mathsf{N-1}} \mathsf{l}[k]\right| > T(b)\right]$, throughout the program. This is difficult to do when there are distributions with infinite support (e.g., the Laplace distribution), or parameters (e.g., the number of samples N). Furthermore, this reasoning must be repeated throughout the proof for each application of the concentration bound. By working with predicates on distributions, we are able to directly model probabilistic independence and concentration bounds, giving a more natural and concise formal proof.

## 3. Programs and assertions

***Programs.*** We base our development on pWhile, a core language with deterministic assignments, probabilistic assignments, conditionals, loops, procedure calls and an $\mathbf{abort}$ statement which halts the computation with no result. Probabilistic assignments are of the form $x \xleftarrow{\$} g$, which assigns a value sampled according to the distribution $g$ to the program variable $x$. The syntax of statements is defined by the grammar:

$$s ::= \mathbf{skip} \mid \mathbf{abort} \mid x \leftarrow e \mid x \xleftarrow{\$} g \mid s; s$$
$$\mid \mathbf{if}\ e\ \mathbf{then}\ s\ \mathbf{else}\ s \mid \mathbf{while}\ e\ \mathbf{do}\ s \mid x \leftarrow \mathcal{F}(e) \mid x \leftarrow \mathcal{A}(e)$$

where $x$, $e$, and $g$ range over (typed) variables in $\mathcal{X}$, expressions in $\mathcal{E}$ and distribution expressions in $\mathcal{D}$ respectively. $\mathcal{E}$ is defined inductively from $\mathcal{X}$ and a set $\mathcal{F}$ of simply typed function symbols, while $\mathcal{D}$ is defined by combining a set of distribution symbols $\mathcal{S}$ with expressions in $\mathcal{E}$. For instance, $e_1 + e_2$ is a valid expression, and $\mathbf{Bern}(e)$—the Bernoulli distribution with parameter $e$—is a valid distribution expression. We assume that expressions, distribution expressions, and statements are typed in the usual way with $\mathbf{SDist}(T)$ the type for probability (sub-)distributions over the type $T$. Ellora can be flexibly extended with custom functions and types.

We distinguish two kinds of procedure calls: $\mathcal{A}$ is a set of external procedure names, and $\mathcal{F}$ is a set of internal procedure names. We assume we have access to the code of internal procedures, but not the code of external procedures. We think of external procedures as controlled by some external *adversary*, who can select the next input in an interactive algorithm. Technically, external procedures run in an external memory separate from the main program memory: We assume that variables $\mathcal{X}$ are partitioned into a set of global variables $\mathcal{X}^{\mathfrak{G}}$, a finite set of local variables $(\mathcal{X}_f^{\mathfrak{L}})_{f\in\mathcal{F}}$, the procedure's formal argument $(\{f_{\mathbf{arg}}\})_{f\in\mathcal{F}}$, and the adversary's state $\{\mathsf{s}\}$. For internal procedures $f \in \mathcal{F}$, we also use the following notations: $f_{\mathbf{body}}$ is the body of the procedure $f$, while $f_{\mathbf{res}} \in \mathcal{E}$ and $f_{\mathbf{init}} = (x \leftarrow e_x)_{x\in\mathcal{X}_f^{\mathfrak{L}}}$ denotes resp. its return statement and the initializers of its local arguments. We assume that the variables occurring in $f_{\mathbf{body}}$ and $f_{\mathbf{res}}$ must be global or local to $f$, while the variables occurring in the expressions of $f_{\mathbf{init}}$ must be global or its formal argument. Moreover, we do not allow recursion in procedures and require that the call-graph of a program is acyclic.

Last, for each adversary $a \in \mathcal{A}$, we denote the type of its formal argument and its return by $a_{\mathbf{argty}}$ and $a_{\mathbf{resty}}$ respectively, and we restrict the set of internal procedures $a$ can call. For that, we associate to $a$ a set $a_{\mathbf{ocl}} \subseteq \mathcal{F}$ representing the internal procedures—known as *oracles*—that $a$ is allowed to call.[1]

***Semantics.*** The denotational semantics of programs is adapted from the seminal work of Kozen [31]. Specifically, we first define the standard interpretation of closed programs as sub-distribution transformers; then, we extend the interpretation to programs with adversary calls. Note that contrary to Kozen [31], who allows general, continuous sub-distributions, our system will work with discrete sub-distributions only to avoid issues of measurability.

**Definition 1.** *A* sub-distribution *over a set $A$ is fully defined by its mass function $\mu : A \to \mathbb{R}^+$ that gives the probability of the unitary events $a \in A$. This mass function must be s.t. $\sum_{a\in A} \mu(a)$ is well-defined and $|\mu| \triangleq \sum_{a\in A} \mu(a) \leq 1$. This implies that the* support $\mathrm{supp}(\mu) \triangleq \{a \in A \mid \mu(a) \neq 0\}$ *is discrete. When the* weight $|\mu|$ *is equal to* 1*, we call $\mu$ a* distribution*. We let $\mathbf{SDist}(A)$ denote the set of sub-distributions over $A$. The probability of an event $E(x)$ w.r.t. a distribution $\mu$, written $\Pr_{x\sim\mu}[E(x)]$, is defined as $\sum_{x|E(x)} \mu(x)$.*

Simple examples of sub-distributions include the *null sub-distribution* $\mathbb{0}^A \in \mathbf{SDist}(A)$, which maps each element of $A$ to 0; and the *Dirac distribution centered on* $x$, written $\delta_x$, which maps $x$ to 1 and all other elements to 0.

We interpret every ground type $T$ as a set $[\![T]\!]$; other constructors $C$ are interpreted as functions $[\![C]\!]$ that respect their arities. The set $\mathbf{State} \triangleq \Pi(x \in \mathcal{X}).\, [\![\tau_x]\!]$ contains all the well-typed maps from variables to values, i.e. all the maps $m$ from $\mathcal{X}$ to $\bigcup_T [\![T]\!]$ s.t. for any $x \in \mathcal{X}$, $m(x) \in [\![\tau_x]\!]$, where $\tau_x$ is the type associated to $x$. For a set of variables $S$, we write $\mathbf{State}(S)$ for the set of all the memories whose domain is $S$ instead of $\mathcal{X}$, and we denote by $m_{|S}$ the element of $\mathbf{State}(S)$ that is obtained by restricting the domain of the memory $m \in \mathbf{State}$ to $S$. One can equip $\mathbf{SDist}(\mathbf{State})$ with a monadic structure, using the Dirac distributions $\delta_x$ for the unit and *distribution expectation* $\mathbb{E}_{x\sim\mu}[M(x)]$ for the bind, where

$$\mathbb{E}_{x\sim\mu}[M(x)] : x \mapsto \sum_a \mu(a) \cdot M(a)(x).$$

The semantics of expressions and distribution expressions is parametrized by a state $m$, and is defined in the usual way where we require all distribution expressions to be interpreted as distributions.

**Definition 2** (Semantics of statements)**.**

- *The semantics $[\![s]\!]_m$ of a closed statement $s$ (i.e. without external procedures) w.r.t. to some initial memory $m$ is a sub-distribution over states, and is defined by the clauses of Figure 2.*
- *The (lifted) semantics $[\![s]\!]_\mu$ of a closed statement $s$ w.r.t. to some initial sub-distribution $\mu$ over memories is a sub-distribution over states, and is defined as $[\![s]\!]_\mu \triangleq \mathbb{E}_{m\sim\mu}[[\![s]\!]_m]$ $\mu \in \mathbf{SDist}(\mathbf{State})$.*

[1] Note that many applications, notably cryptography, restrict adversaries in terms of how many oracle calls they may perform. These restrictions can also be accommodated in Ellora, although we do not develop them further here.

- *The semantics $[\![s]\!]_m$ of a statement $s$ with external procedures w.r.t. to some initial memory $m$ is defined relative to an external procedures context $\rho$, and is given by the clause*

$$[\![x \leftarrow a(e)]\!]_{\rho,m} \triangleq [\![a_{\mathbf{arg}} \leftarrow e; \rho_{\mathsf{body}}(a); x \leftarrow \rho_{\mathsf{res}}(a)]\!]_m$$

*where an external procedures context $\rho$ is a pair of mappings $\rho_{\mathsf{body}}$ and $\rho_{\mathsf{res}}$ s.t. for $a \in \mathcal{F}$, $\rho_{\mathsf{body}}(a)$ is a closed statement that only makes internal procedures calls to procedures in $a_{\mathbf{ocl}}$, $\rho_{\mathsf{res}}(a)$ is a return expression and the only variables occurring in $\rho_{\mathsf{body}}(a)$ or $\rho_{\mathsf{res}}(a)$ are $a_{\mathbf{arg}}$ or $\mathtt{s}$. Moreover, we assume the adversary code to be in normal form w.r.t. the oracle calls: Its body must be of the form $s_1; c_1^?; \cdots; s_n; c_n^?$ where the $s_i$'s are commands without calls to the oracles, whereas the $c_i^?$'s are potential calls to the oracles.*

The semantics definition (Fig. 2) is mostly standard; the most interesting case is for loops, where the interpretation of a **while** loop is the limit of the interpretations of its finite unrollings. Formally, the $n^{th}$ *truncated iterate* of the loop $\mathbf{while}\ b\ \mathbf{do}\ s$ is defined as

$$\overbrace{\mathbf{if}\ b\ \mathbf{then}\ s; \ldots; \mathbf{if}\ b\ \mathbf{then}\ s}^{n \text{ times}}; \mathbf{if}\ b\ \mathbf{then\ abort}$$

which we represent using the shorthand $(\mathbf{if}\ b\ \mathbf{then}\ s)^n_{|\neg b}$. For any initial sub-distribution $\mu$, applying the trunacted iterates yields an increasing and bounded sequence of sub-distributions. We take the limit of this sequence to give a semantics to the **while** loop. Moreover, it is well-known from Kozen [31] that the semantics of **while** loops can also be defined as the fixed point of the operator

$$\mu \mapsto [\![\mathbf{if}\ b\ \mathbf{then}\ s]\!]_\mu.$$

Kozen [31] shows that this is well-defined by applying Banach's fixed point theorem. The definition also satifies the equation

$$[\![\mathbf{while}\ b\ \mathbf{do}\ s]\!]_\mu = [\![\mathbf{if}\ b\ \mathbf{then}\ s; \mathbf{while}\ b\ \mathbf{do}\ s]\!]_\mu.$$

We make use of the following observation to define sound proof rules for **while** loops.

**Lemma 3.** *The sequence $[\![(\mathbf{if}\ b\ \mathbf{then}\ s)^n]\!]_\mu$ has a limit, provided $|[\![\mathbf{while}\ b\ \mathbf{do}\ s]\!]_\mu| = 1$. In this case, we also have*

$$\lim_{n\to\infty} [\![(\mathbf{if}\ b\ \mathbf{then}\ s)^n]\!]_\mu = [\![\mathbf{while}\ b\ \mathbf{do}\ s]\!]_\mu.$$

*Proof.* It suffices to consider the statement for a memory $m$. The sequence $[\![(\mathbf{if}\ b\ \mathbf{then}\ s)^n]\!]_m$ is bounded below by the sequence $[\![(\mathbf{if}\ b\ \mathbf{then}\ s)^n_{|\neg b}]\!]_m$. Since the latter converges to a distribution, the sequence $[\![(\mathbf{if}\ b\ \mathbf{then}\ s)^n]\!]_m$ also admits a limit, and moreover it converges to the same limit as $[\![(\mathbf{if}\ b\ \mathbf{then}\ s)^n_{|\neg b}]\!]_m$, i.e. $[\![\mathbf{while}\ b\ \mathbf{do}\ s]\!]_m$. □

We conclude this section with a taxonomy of the termination behavior of statements and loops.

**Definition 4** (Lossless)**.** *A statement $s$ is* lossless *iff for every memory $m$, $|[\![s]\!]_m| = 1$.*

If $s$ is lossless, then for every sub-distribution $\mu$, $|[\![s]\!]_\mu| = |\mu|$; this follows directly from additivity of the semantics. If $s$ is not lossless, then we still have $|[\![s]\!]_\mu| \leq |\mu|$. Programs that are not lossless are called *lossy*.

**Definition 5** (Certain and almost sure termination)**.** *A loop $\mathbf{while}\ b\ \mathbf{do}\ s$ is:*

- certainly (c.) terminating *if there exists $n$ such that for every sub-distribution $\mu$: $|[\![\mathbf{while}\ b\ \mathbf{do}\ s]\!]_\mu| = |[\![(\mathbf{if}\ b\ \mathbf{then}\ s)^n_{|\neg b}]\!]_\mu|$.*
- almost surely (a.s.) terminating *if it is lossless.*

Note that, if $\mathbf{while}\ b\ \mathbf{do}\ s$ is a.s. terminating, then for every sub-distribution $\mu$, $[\![\mathbf{while}\ b\ \mathbf{do}\ s]\!]_\mu = \lim_{n\to\infty} [\![(\mathbf{if}\ b\ \mathbf{then}\ s)^n]\!]_\mu$ by Lemma 3. Certain termination is similar to termination in deterministic programs, whereas almost sure termination is probabilistic in nature: the program always terminates eventually, but we may not be able to give a single finite bound for all executions since particular executions may proceed arbitrarily long. Note that certain termination need not entail losslessness.

***Assertions.*** We model assertions as predicates on states.

**Definition 6** (Assertions and satisfaction)**.** *The set* Assn *of assertions is defined as $\mathcal{P}(\mathbf{SDist}(\mathbf{State}))$. We write $\eta(\mu)$ for $\mu \in \eta$.*

Usual set operations are lifted to assertions using their logical counterparts, e.g., $\eta \wedge \eta' \triangleq \eta \cap \eta'$ and $\neg\eta \triangleq \overline{\eta}$. Beside these standard constructions, we frequently use the following assertions. Given a predicate $\phi$ over states, we let $\Box\phi$ be the assertion defined by:

$$\Box\phi \triangleq \lambda\mu.\, \forall m.\, m \in \mathrm{supp}(\mu) \implies \phi(m)$$

Intuitively, this means that $\phi$ holds on all memories with that can be drawn from the distribution. Moreover, given two assertions $\eta_1$ and $\eta_2$, we let $\eta_1 \oplus \eta_2$ be the assertion defined by the clause:

$$\eta_1 \oplus \eta_2 \triangleq \lambda\mu.\, \exists\mu_1, \mu_2.\, \mu = \mu_1 + \mu_2 \wedge \eta_1(\mu_1) \wedge \eta_2(\mu_2)$$

Intuitively, this assertion means that the sub-distribution is the sum of two sub-distributions, such that $\eta_1$ holds on the first piece and $\eta_2$ holds on the second piece. Finally, given an assertion $\eta$ and a function $F$ from $\mathbf{SDist}(\mathbf{State})$ to $\mathbf{SDist}(\mathbf{State})$, we let $\eta[F]$ be the assertion defined by the clause: $\eta[F] \triangleq \lambda\mu.\, \eta(F(\mu))$.

Now, we define the closedness properties of assertions. These properties will later be used to achieve soundness of the rules for **while** loops.

**Definition 7** (Closedness properties)**.**

- *An assertion $\eta$ is* $t$-closed *if for every converging sequence of sub-distributions $(\mu_n)_{n\in\mathbb{N}}$ such that $\eta(\mu_n)$ for all $n \in \mathbb{N}$ then*

$$\eta(\lim_{n\to\infty} \mu).$$

- *An assertion $\eta$ is* $d$-closed *if it is $t$-closed and downward closed, that is for every sub-distributions $\mu \leq \mu'$, $\eta(\mu')$ implies $\eta(\mu)$.*

Both properties have a topological interpretation as closedness under the $L^1$-topology and the $L^1$- and upper-topology, respectively.

**Proposition 8.**

1. *If $\eta$ is $d$-closed and is s.t.*

$$\forall\mu.\, \mu \models \eta \implies [\![\mathbf{if}\ b\ \mathbf{then}\ s]\!]_\mu \models \eta,$$

*then $\forall\mu.\, \mu \models \eta \implies [\![\mathbf{while}\ b\ \mathbf{do}\ s]\!]_\mu \models \eta$.*

2. *If $\eta$ is $t$-closed and is s.t.*

$$\forall\mu.\, \mu \models \eta \implies [\![\mathbf{if}\ b\ \mathbf{then}\ s]\!]_\mu \models \eta,$$

*then $\forall\mu.\, \mu \models \eta \implies [\![\mathbf{while}\ b\ \mathbf{do}\ s]\!]_\mu \models \eta$, provided that $\mathbf{while}\ b\ \mathbf{do}\ s$ is lossless.*

*Proof.* We only treat the first case; the second is similar. Let $\eta$ be a $d$-closed assertion s.t. for any sub-distribution $\mu$, if $\eta(\mu)$, then $[\![\mathbf{if}\ b\ \mathbf{then}\ s]\!]_\mu \models \eta$. We prove by induction on $n$ that for any sub-distribution $\mu$ such that $\mu \models \eta$, we have $[\![(\mathbf{if}\ b\ \mathbf{then}\ s)^n]\!]_\mu \models \eta$. By downward closedness of $\eta$, we have $[\![(\mathbf{if}\ b\ \mathbf{then}\ s)^n_{|\neg b}]\!]_\mu \models \eta$. We conclude by $t$-closenedness of $\eta$. □

While the closedness is a semantic property, there are several sufficient conditions that are easier to check. First, both $t$-closed and $d$-closed assertions are closed under finite boolean combinations, universal quantification over arbitrary sets and existential quantification over finite sets. We can give some examples:

$$
\begin{aligned}
[\![\mathbf{skip}]\!]_m &= \delta_m & [\![\mathbf{abort}]\!]_m &= \mathbb{0} \\
[\![x \leftarrow e]\!]_m &= \delta_{m[x:=[\![e]\!]_m]} & [\![x \xleftarrow{\$} g]\!]_m &= \mathbb{E}_{v\sim[\![g]\!]m}[\delta_{m[x:=v]}] \\
[\![s_1; s_2]\!]_m &= \mathbb{E}_{\xi\sim[\![s_1]\!]_m}[[\![s_2]\!]_\xi] & [\![\mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2]\!]_m &= \text{if } [\![e]\!]_m \text{ then } [\![s_1]\!]_m \text{ else } [\![s_2]\!]_m \\
[\![x \leftarrow f(e)]\!]_m &= \mathbb{E}_{\xi\sim[\![f_{\mathbf{arg}}:=e;f_{\mathbf{init}};f_{\mathbf{body}}]\!]_m}[\delta_{m[x:=[\![f_{\mathbf{res}}]\!]_\xi]}] & [\![\mathbf{while}\ b\ \mathbf{do}\ s]\!]_m &= \lim_{n\to\infty} [\![(\mathbf{if}\ b\ \mathbf{then}\ s)^n_{|\neg b}]\!]_m
\end{aligned}
$$

**Figure 2.** Denotational semantics of programs

- assertions of the form $p_1 \bowtie p_2$, where $\bowtie$ is a non-strict comparison operator ($\bowtie \in \{\leq, \geq, =\}$) for bounded probabilistic expressions $p_1, p_2$—for instance probabilities or expectations of bounded variables—are $t$-closed. There are simple examples where $t$-closedness fails for unbounded expressions. An example of a $t$-closed assertion is the equivalence of two variables $x, y$:

$$\overbrace{\forall n \in \mathbb{N},}^{\text{Universal quantification}} \quad \overbrace{\Pr[x = n]}^{\text{Bounded expression}} = \overbrace{\Pr[y = n]}^{\text{Bounded expression}}$$

- assertions of the form $p_1 \leq k$ for bounded probabilistic expression $p_1$ are $d$-closed.

## 4. Proof system

In this section, we introduce a program logic for proving properties of probabilistic programs, and prove its soundness.

***Judgments and proof rules.*** Judgments are of the form $\{\eta\}\ s\ \{\eta'\}$, where $\eta, \eta' \in \mathsf{Assn}$.

**Definition 9.** *A judgment $\{\eta\}\ s\ \{\eta'\}$ is* valid*, written $\models \{\eta\}\ s\ \{\eta'\}$, if $\eta'([\![s]\!]_\mu)$ for every probabilistic state $\mu$ such that $\eta(\mu)$.*

Figure 3 describes the structural and basic rules of the proof system. Validity of judgments is preserved under standard structural rules, like the rule of consequence [CONSEQ]. The rule of consequence is especially important for our purposes, since it serves as the interface between the program logic and theorems probability theory. For instance, this rule is how we can apply concentration bounds and other mathematical results.

The rules for **skip**, assignments, random samplings and sequences are all straightforward. The rule for **abort** requires $\square\bot$ to hold after execution; this assertion uniquely characterizes the resulting null sub-distribution. The rules for assignments and random samplings are semantical; we give more concrete versions in the next section. The [CALL] rule for procedure calls reduces to proving the given pre- and post-conditions on the body of the procedure.

The rule [COND] for conditionals is unusual in that the post-condition must be of the form $\eta_1 \oplus \eta_2$; this reflects the semantics of conditionals, which splits the initial probabilistic state depending on the guard, runs both branches, and adds the resulting two probabilistic states.

The next two rules ([SPLIT] and [FRAME]) are critical for local reasoning. The [SPLIT] rule reflects the additivity of the semantics and can be used for simultaneously recombining pre- and post-conditions using the $\oplus$ operator. The [FRAME] rule states that lossless statements preserve assertions that are not influenced by its set $\mathrm{mod}(s)$ of *modified* variables: the variables on the left of an assignment, a random sampling or a procedure call. In this setting, we say that an assertion $\eta$ is *separated* from a set of variables $X$, written $\mathsf{separated}(\eta, X)$, if $\eta(\mu_1) \iff \eta(\mu_2)$ for any distributions $\mu_1, \mu_2$ s.t. $|\mu_1| = |\mu_2|$ and $\mu_{1|\overline{X}} = \mu_{2|\overline{X}}$ where for a set $S$, $\mu_{|S}$ is defined as:

$$\mu_{|S} : m \in \mathbf{State}_{|S} \mapsto \Pr_{m'\sim\mu}[m = m'_{|S}]$$

$$\textsc{Conseq}\ \frac{\eta_0 \Rightarrow \eta_1 \qquad \{\eta_1\}\ s\ \{\eta_2\} \qquad \eta_2 \Rightarrow \eta_3}{\{\eta_0\}\ s\ \{\eta_3\}} \qquad \textsc{Abort}\ \frac{}{\{\eta\}\ \mathbf{abort}\ \{\square\bot\}}$$

$$\textsc{Assgn}\ \frac{\eta' \triangleq \eta[[\![x \leftarrow e]\!]]}{\{\eta'\}\ x \leftarrow e\ \{\eta\}} \qquad \textsc{Sample}\ \frac{\eta' \triangleq \eta[[\![x \xleftarrow{\$} g]\!]]}{\{\eta'\}\ x \xleftarrow{\$} g\ \{\eta\}} \qquad \textsc{Skip}\ \frac{}{\{\eta\}\ \mathbf{skip}\ \{\eta\}}$$

$$\textsc{Seq}\ \frac{\{\eta_0\}\ s_1\ \{\eta_1\} \qquad \{\eta_1\}\ s_2\ \{\eta_2\}}{\{\eta_0\}\ s_1; s_2\ \{\eta_2\}}$$

$$\textsc{Cond}\ \frac{\{\eta_1 \wedge \square e\}\ s_1\ \{\eta'_1\} \qquad \{\eta_2 \wedge \square\neg e\}\ s_2\ \{\eta'_2\}}{\{(\eta_1 \wedge \square e) \oplus (\eta_2 \wedge \square\neg e)\}\ \mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2\ \{\eta'_1 \oplus \eta'_2\}}$$

$$\textsc{Split}\ \frac{\{\eta_1\}\ s\ \{\eta'_1\} \qquad \{\eta_2\}\ s\ \{\eta'_2\}}{\{\eta_1 \oplus \eta_2\}\ s\ \{\eta'_1 \oplus \eta'_2\}}$$

$$\textsc{Call}\ \frac{\{\eta\}\ f_{\mathbf{arg}} \leftarrow e; f_{\mathbf{init}}; f_{\mathbf{body}}\ \{\eta'[[\![x \leftarrow f_{\mathbf{res}}]\!]]\}}{\{\eta\}\ x \leftarrow f(e)\ \{\eta'\}}$$

$$\textsc{Frame}\ \frac{\mathsf{separated}(\eta, \mathrm{mod}(s)) \qquad s \text{ is lossless}}{\{\eta\}\ s\ \{\eta\}}$$

**Figure 3.** Structural and basic rules

Intuitively, an assertion is separated from a set of variables $X$ if every two sub-distributions that agree on the variables outside $X$ either both satisfy the assertion, or both refute the assertion.

Figure 4 presents the rules for **while** loops and external procedure calls. The [WHILE] rule has three instantiations, depending on the termination behavior of the loop. As usual, we must provide a loop invariant; in our case, a loop invariant is an arbitrary assertion that is preserved by one (guarded) iteration of the loop. The instantiations consider arbitrary, almost surely, and certainly terminating loops. In the general case, when no restriction are required about the termination behavior, we require the invariant to be $d$-closed. This condition can be weakened for terminating loops: in the case of a loop that terminates surely, a $t$-closed invariant is sufficient; whereas we do not require anything for loops terminating certainly.

The rule [ADV] for external procedure calls follows the same idea: it states when an assertion that is preserved by oracle calls is also preserved by the external procedure call. Some framing conditions are required, similar to the ones of the [FRAME] rule: the invariant must not be influenced by the state writable by the external procedures, which also must be lossless. Similar to the loop rule, different flavors exist. In Figure 4, we only give the most general one where the invariant is required to be $d$-closed. However, for

example, this restriction can be removed by bounding the number of calls the external procedure can make to oracles, leading to a rule akin to the certain termination case of the loop rule.

***Soundness.*** Our proof system is sound w.r.t. the semantics.

**Theorem 10** (Soundness)**.** *Every judgment* $\{\eta\}\ s\ \{\eta'\}$ *provable using the rules of our logic is valid.*

Completeness of the logic is left for future work.

## 5. A two-level syntax for assertions

So far, we have seen a version of Ellora where assertions are arbitrary predicates on distributions. While this version is quite general, it is desirable for practical applications to define a syntax for assertions and to develop specialized proof rules which are easier to use.

***Assertions.*** Fig. 5 presents a two-level assertion language which is sufficiently expressive for typical assertions that arise in the analysis of randomized algorithms. The assertion language supports probabilistic assertions and state assertions. A *probabilistic assertion* $\eta$ is a formula built from comparison of probabilistic expressions, using first-order quantifiers and connectives, and the special connective $\oplus$. A *probabilistic expression* $p$ can be a logical variable $v$, an operator applied to probabilistic expressions $o(\vec{p})$ (constants are 0-ary operators), or the expectation $\mathbb{E}[\tilde{e}]$ of a state expression $\tilde{e}$. A *state expression* $\tilde{e}$ is either a program variable $x$, the characteristic function $\mathbf{1}_\phi$ of a state assertion $\phi$, an operator applied to state expressions $o(\vec{\tilde{e}})$, or the expectation $\mathbb{E}_{v\sim g}[\tilde{e}]$ of state expression $\tilde{e}$ in a given distribution $g$. Finally, a *state assertion* $\phi$ is a first-order formula over program variables. Note that the set of operators is left unspecified but we assume that all the expressions in $\mathcal{E}$ and $\mathcal{D}$ can be encoded by operators.

The interpretation of the concrete syntax is as expected. The interpretation of probabilistic assertions is relative to a valuation $\rho$ which maps logical variables to values, and is an element of Assn. The definition of the interpretation is straightforward; the only interesting case is $[\![\mathbb{E}[\tilde{e}]]\!]^\rho_\mu$ which is defined by $\mathbb{E}_{m\sim\mu}[[\![\tilde{e}]\!]^\rho_m]$, where $[\![\tilde{e}]\!]^\rho_m$ is the interpretation of the state expression $\tilde{e}$ in the memory $m$ and valuation $\rho$. The interpretation of state expressions is a mapping from memories to values, which can be lifted to a mapping from distributions over memories to distributions over values. The definition of the interpretation is straightforward; the most interesting case is for expectation $[\![\mathbb{E}_{v\sim g}[\tilde{e}]]\!]^\rho_m \triangleq \mathbb{E}_{w\sim[\![g]\!]^\rho_m}[[\![\tilde{e}]\!]^{\rho[v:=w]}_m]$.

Many standard concepts from probability theory have a natural representation in our syntax. We give some examples:

- the probability that $\phi$ holds in some probabilistic state is represented by the probabilistic expression $\Pr[\phi] \triangleq \mathbb{E}[\mathbf{1}_\phi]$;
- probabilistic independence of state expressions $\tilde{e}_1, \ldots, \tilde{e}_n$ is modeled by the probabilistic assertion $\#\{\tilde{e}_1, \ldots, \tilde{e}_n\}$, defined by the clause[2]

$$\forall v_1 \ldots v_n,\ \Pr[\top]^{n-1} \Pr[\bigwedge_{i=1\ldots n} \tilde{e}_i = v_i] = \prod_{i=1\ldots n} \Pr[\tilde{e}_i = v_i];$$

- losslessness of a distribution is modeled by the probabilistic assertion $\mathcal{L} \triangleq \Pr[\top] = 1$;
- a state expression $\tilde{e}$ distributed according to a law $g$ is modeled by the probabilistic assertion $\tilde{e} \sim g$ defined as:

$$\forall w,\ \Pr[\tilde{e} = w] = \mathbb{E}[\mathbb{E}_{v\sim g}[\mathbf{1}_{v=w}]].$$

The inner expectation computes the probability that $v$ drawn from $g$ is equal to a fixed $w$; the outer expectation weights the inner probability by the probability of each value of $w$.

We can easily define $\Box$ operator from the previous section in our new syntax: $\Box\phi \triangleq \Pr[\neg\phi] = 0$.

***Syntactic proof rules.*** Now that we have a concrete syntax for assertions, we can give syntactic versions of many of the existing proof rules. Such proof rules are often easier to use, since they avoid reasoning about the semantics of commands and assertions. We tackle the non-looping rules first, beginning with the following syntactic rules for assignment and sampling:

$$\textsc{Sample}\quad \frac{}{\{\eta[x := e]\}\ x \leftarrow e\ \{\eta\}} \qquad \textsc{Assgn}\quad \frac{}{\{\mathcal{P}^g_x(\eta)\}\ x \xleftarrow{\$} g\ \{\eta\}}$$

The rule for assignment is the usual rule from Hoare logic, replacing the program variable $x$ by its corresponding expression $e$ in the pre-condition. The replacement $\eta[x := e]$ is done recursively on the probabilistic assertion $\eta$; for expectation it is defined by

$$\mathbb{E}[\tilde{e}][x := e] \triangleq \mathbb{E}[\tilde{e}[x := e]]$$

where $\tilde{e}[x := e]$ is the syntactic substitution of the variable $x$ by the expression $e$ in $\tilde{e}$.

The rule for sampling is a generalization of assignment using a probabilistic substitution operator $\mathcal{P}^g_x(\eta)$, which replaces all occurrences of $x$ in $\eta$ by a new integration variable $t$ and records that $t$ is drawn from $g$. More formally, $\mathcal{P}^g_x(\eta)$ is defined recursively on probabilistic assertions; for expectation, it is defined as

$$\mathcal{P}^g_x\left(\mathbb{E}[\tilde{e}]\right) \triangleq \mathbb{E}[\mathbb{E}_{t\sim g}[\tilde{e}[x := t]]].$$

Next, we turn to the loop rule. The side conditions from Fig. 4 are purely semantic, while in practice it is more convenient to use a sufficient condition in the Hoare logic. We give sufficient conditions for ensuring certain and almost-sure termination in Fig. 6.

The first side condition $\mathcal{C}_{\text{CTerm}}$ shows certain termination given a strictly decreasing *variant* $\tilde{e}$ that is bounded below, similar to how a decreasing variant shows termination for deterministic programs. The second side condition $\mathcal{C}_{\text{ASTerm}}$ shows almost-sure termination given a probabilistic variant $\tilde{e}$, which must be bounded both above and below. While $\tilde{e}$ may increase with some probability, it must decrease with strictly positive probability. This sufficient condition for almost-sure termination was previously considered by Hart et al. [19] for probabilistic transition systems, and also used in expectation-based approaches [23, 38].

***Precondition calculus.*** With a concrete syntax for assertions, we are also able to incorporate syntactic reasoning principles. One classic tool is Morgan and McIver's *greatest pre-expectation*, which we take as inspiration for a pre-condition calculus for the loop-free fragment of Ellora. Given an assertion $\eta$ and a loop-free statement $s$, we wish to mechanically construct an assertion $\eta^*$ that is the pre-condition of $s$ that implies $\eta$ as a post-condition.

Given a statement $s$ and a probabilistic assertion $\eta$, the computation of the pre-condition replaces each expectation expression $p$ inside $\eta$ by an expression $p^*$ that has the same denotation before running $s$ as $p$ after running $s$. This process yields an assertion $\eta^*$ that, interpreted before running $s$, is logically equivalent to $\eta$ interpreted after running $s$.

The computation rules for pre-conditions are defined in Fig. 7. For a probability assertion $\eta$, its pre-condition $\text{pc}(s, \eta)$ corresponds to $\eta$ where the expectation expressions of the form $\mathbb{E}[\tilde{e}]$ are replaced by their corresponding *preterm*, $\text{pe}(s, \mathbb{E}[\tilde{e}])$. Preterms correspond loosely to Morgan and McIver's *pre-expectations*—we will make this correspondence more precise in the next section. The main interesting cases for computing preterms are for random sampling

[2] The term $\Pr[\top]^{n-1}$ is necessary since we work with sub-distributions; for distributions, $\Pr[\top] = 1$ and we recover the usual definition.

$$\textsc{While-X}\ \frac{\{\eta\}\ \mathbf{if}\ b\ \mathbf{then}\ s\ \{\eta\} \qquad \mathcal{C}_{\mathrm{X}}}{\{\eta\}\ \mathbf{while}\ b\ \mathbf{do}\ s\ \{\eta \wedge \square\neg b\}} \qquad \textsc{Adv}\ \frac{\mathsf{separated}(\eta, \{x, \mathfrak{s}\}) \qquad \eta \text{ is } d\text{-closed} \qquad a \text{ is lossless} \qquad \forall f \in a_{\mathbf{ocl}}, x \in \mathcal{X}_a^{\mathfrak{L}}, e \in \mathcal{E}.\ \{\eta\}\ x \leftarrow f(e)\ \{\eta\}}{\{\eta\}\ x \leftarrow a(e)\ \{\eta\}}$$

SIDE CONDITIONS:

$$\mathcal{C}_{\mathrm{CTerm}} \triangleq \forall \mu.\, \eta(\mu) \implies \exists k \in \mathbb{N}.\, \Pr_{m\sim\nu}[[\![b]\!]_m] = 0 \qquad \text{where } \nu \triangleq [\![(\mathbf{if}\ b\ \mathbf{then}\ s)^k]\!]_\mu$$

$$\mathcal{C}_{\mathrm{ASTerm}} \triangleq \eta\ t\text{-closed and } \forall \mu.\, \eta(\mu) \implies |\nu| = 1 \qquad \text{where } \nu \triangleq [\![\mathbf{while}\ b\ \mathbf{do}\ s]\!]_\mu$$

$$\mathcal{C}_{\mathrm{ATerm}} \triangleq \eta\ d\text{-closed}$$

**Figure 4.** Rules for **while** loops and external calls

$$\begin{aligned}
\tilde{e} &::= x \mid v \mid \mathbf{1}_\phi \mid \mathbb{E}_{v\sim g}[\tilde{e}] \mid o(\vec{\tilde{e}}) && \text{(S-expr.)}\\
\phi &::= \tilde{e} \bowtie \tilde{e} \mid FO(\phi) && \text{(S-assn.)}\\
p &::= v \mid o(\vec{p}) \mid \mathbb{E}[\tilde{e}] && \text{(P-expr.)}\\
\eta &::= p \bowtie p \mid \eta \oplus \eta \mid FO(\eta) && \text{(P-assn.)}\\
\bowtie &\in \{=, <, \leq\} \qquad o \in Ops && \text{(Operators)}
\end{aligned}$$

**Figure 5.** Assertion syntax

and conditionals. For random sampling the result is $\mathcal{P}_x^g(\mathbb{E}[\tilde{e}])$, which corresponds to the [SAMPLE] rule. For conditionals, the expectation expression is split into a part where $e$ is true and a part where $e$ is not true. We restrict the expectation to a part satisfying $e$ with the following operator:

$$\mathbb{E}[\tilde{e}]_{|e} \triangleq \mathbb{E}[\tilde{e} \cdot \mathbf{1}_e]$$

This corresponds to the expected value of $\tilde{e}$ on the portion of the distribution where $e$ is true.

Then, we can build the pre-condition calculus into Ellora.

**Theorem 11.** *Let $s$ be a non-looping command. Then, the following rule is derivable in the concrete version of Ellora:*

$$\text{PC}\ \frac{}{\{pc(s,\eta)\}\ s\ \{\eta\}}$$

## 6. Embedding logics

While our presentation of Ellora so far is suitable for general-purpose reasoning about probabilistic programs, in practice proofs typically use more lightweight, specific reasoning principles to prove certain assertions. To see that such patterns can also be naturally used in Ellora, we consider embeddings of three tools in our framework: the union bound logic from Barthe et al. [7], a fragment of pGCL [39], and a new logic for reasoning about independence.

***Union bound logic.*** Barthe et al. [7] have recently introduced a lightweight program logic, called aHL, for estimating accuracy of randomized computations. One main application of aHL is proving accuracy of randomized algorithms, both in the offline and online settings—i.e. with adversary calls. aHL is based on the union bound, a basic tool from probability theory, and has judgments of the form

$$\models_\beta \{\Phi\}\ s\ \{\Psi\},$$

where $s$ is a statement, $\Phi$ and $\Psi$ are first-order formulae over program variables, and $\beta$ is a probability, i.e. $\beta \in [0, 1]$. A judgment $\models_\beta \{\Phi\}\ s\ \{\Psi\}$ is valid if for every memory $m$ such that $\Phi(m)$, the probability of $\neg\Psi$ in $[\![s]\!]_m$ is upper bounded by $\beta$, i.e. $\Pr_{[\![s]\!]_m}[\neg\Psi] \leq \beta$.

Figure 8 presents some key rules of aHL, including a rule for sampling from the Laplace distribution $\mathcal{L}_\epsilon$ centered around $e$. The predicate $\mathcal{C}_{\mathrm{CTerm}}(k)$ indicates that the loop terminates in at most $k$ steps on any memory that satisfies the pre-condition. Moreover, $\beta$ is a function of $\epsilon$. aHL has a simple embedding into Ellora.

**Theorem 12** (Embedding of aHL)**.** *If $\models_\beta \{\Phi\}\ s\ \{\Psi\}$ is derivable, then $\{\square\Phi\}\ s\ \{\mathbb{E}[\mathbf{1}_{\neg\Psi}] \leq \beta\}$ is derivable in Ellora.*

Of course, there are valid judgments that cannot be derived with the aHL rules but are provable using Ellora. For instance, the proof of the private sum example from § 2 requires applying concentration bounds and reasoning about independence, which are not supported in aHL.

***The logic pGCL.*** Probabilistic Guarded Command Language (pGCL) [39] is a well-studied deductive system for reasoning about programs. This language is the same as the core imperative language that we consider, except instead of a random sampling command from general distributions $x \xleftarrow{\$} g$, pGCL encodes random choice with a probabilistic guarded command of the form $s_1 \oplus_p s_2$. This command executes $s_1$ with probability $p$, otherwise it executes $s_2$. For the embedding, we will simulate this command by sampling from the Bernoulli (coin flip) distribution with parameter $p$, denoted $B(p)$. pGCL also notably supports reasoning about various kinds of non-deterministic choice; we do not consider these features here. We also will only embed loop-free pGCL commands.

The key object in pGCL reasoning is an *expectation*—a map from states to real numbers—and the key tool is *greatest pre-expectation*. This procedure takes a command $s$ and an expectation $E$, and mechanically computes an expectation $\mathrm{gpe}(s, E)$ such that if we view $s$ as a map from an input memory $m$ to an output distribution $\mu(m)$, then

$$\mathrm{gpe}(s, E)(m) = \mathbb{E}_{s'\sim\mu(m)}[E]$$

for every memory $m$. That is, the greatest pre-expectation takes an expectation $E$ on the *output* distribution, and transforms it into an expectation $E'$ that takes the same value as $E$ when $E'$ is evaluated on the *input* memory. In this way, we can calculate a target final expectation by propagating it backwards through the program, until we compute a mathematical formula for the expectation as a function of the input memory only.

For the embedding, we will assume that each expectation $E$ can be directly interpreted as a state expression $\tilde{e}$. The embedding uses the precondition calculus presented in the previous section. We will need one technical lemma.

**Lemma 13.** *For every state expression $\tilde{e}$, loop-free and deterministic pGCL program $s$, and real number $\alpha$,*

$$\square(gpe(s, \tilde{e}) = \alpha) \implies pc(\lceil s \rceil, \mathbb{E}[\tilde{e}] = \alpha)$$

$$\mathcal{C}_{\text{CTerm}} \triangleq \{\mathcal{L} \wedge \square(\tilde{e} = k \wedge 0 < k \wedge b)\}\ s\ \{\mathcal{L} \wedge \square(\tilde{e} < k)\} \quad \tilde{e} : \mathbb{N}$$
$$\models \eta \Rightarrow (\exists \dot{y}.\ \square \tilde{e} \leq \dot{y}) \wedge \square(\tilde{e} = 0 \Rightarrow \neg b)$$

$$\mathcal{C}_{\text{ASTerm}} \triangleq \{\mathcal{L} \wedge \square(\tilde{e} = k \wedge 0 < k \leq K \wedge b)\}\ s\ \{\mathcal{L} \wedge \square(0 \leq \tilde{e} \leq K) \wedge \Pr[\tilde{e} < k] \geq \epsilon\} \quad \tilde{e} : \mathbb{N}$$
$$\models \eta \Rightarrow \square(0 \leq \tilde{e} \leq K \wedge \tilde{e} = 0 \Rightarrow \neg b)$$
$$\models \eta\ \ t\text{-closed}$$

**Figure 6.** Side-conditions for loop rules

$$\begin{aligned}
&\mathrm{pe}(s_1; s_2, \mathbb{E}[\tilde{e}]) &&\triangleq \mathrm{pe}(s_1, \mathrm{pe}(s_2, \mathbb{E}[\tilde{e}]))\\
&\mathrm{pe}(x \leftarrow e, \mathbb{E}[\tilde{e}]) &&\triangleq \mathbb{E}[\tilde{e}][x := e]\\
&\mathrm{pe}(x \xleftarrow{\$} g, \mathbb{E}[\tilde{e}]) &&\triangleq \mathcal{P}^g_x(\mathbb{E}[\tilde{e}])\\
&\mathrm{pe}(\textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2, \mathbb{E}[\tilde{e}]) &&\triangleq \mathrm{pe}(s_1, \mathbb{E}[\tilde{e}])_{|e} + \mathrm{pe}(s_2, \mathbb{E}[\tilde{e}])_{|\neg e}\\
&\mathrm{pc}(s, p_1 \bowtie p_2) &&\triangleq \mathrm{pe}(s, p_1) \bowtie \mathrm{pe}(s, p_2)
\end{aligned}$$

**Figure 7.** Precondition calculus (selected)

$$\frac{}{\models_\beta \{\top\}\ x \xleftarrow{\$} \mathcal{L}_\epsilon(e)\ \{|x - e| \leq \frac{1}{\epsilon} \log \frac{1}{\beta}\}}$$

$$\frac{\models_{\beta_1} \{\Phi\}\ s_1\ \{\Theta\} \qquad \models_{\beta_2} \{\Theta\}\ s_2\ \{\Psi\}}{\models_{\beta_1+\beta_2} \{\Phi\}\ s_1; s_2\ \{\Psi\}}$$

$$\frac{\models_\beta \{\Phi\}\ c\ \{\Phi\} \qquad \mathcal{C}_{\text{CTerm}}(k)}{\models_{k\cdot\beta} \{\Phi\}\ \textbf{while } e \textbf{ do } c\ \{\Phi \wedge \neg e\}}$$

**Figure 8.** aHL proof rules (selected)

*where $\lceil s \rceil$ is the program obtained from $s$ by replacing all occurrences of the probabilistic choice $s_1 \oplus_p s_2$ by $x \xleftarrow{\$} B(p); \textbf{if } x \textbf{ then } s_1 \textbf{ else } s_2$ for a fresh variable $x$.*

Then, we can embed a fragment of pGCL into Ellora.

**Theorem 14** (Embedding of core pGCL)**.** *For every state expression $\tilde{e}$, loop-free and deterministic pGCL program $s$, and real number $\alpha$, the following judgment is derivable in Ellora:*

$$\{\square(gpe(s, \tilde{e}) = \alpha)\}\ \lceil s \rceil\ \{\mathbb{E}[\tilde{e}] = \alpha\}.$$

*Proof.* Since $\lceil s \rceil$ is loop-free, we can derive

$$\{\mathrm{pc}(\lceil s \rceil, \mathbb{E}[\tilde{e}] = \alpha)\}\ \lceil s \rceil\ \{\mathbb{E}[\tilde{e}] = \alpha\}.$$

by rule [PC]. By Lemma 13 we also have

$$\square(\mathrm{gpe}(s, \tilde{e}) = \alpha) \implies \mathrm{pc}(\lceil s \rceil, \mathbb{E}[\tilde{e}] = \alpha),$$

so we can conclude with the rule [CONSEQ]. □

***Law and Independence Logic.*** Our final example is a proof system for reasoning about probabilistic independence and distribution laws. This type of reasoning is common when analyzing randomized algorithms; yet, it is particularly hard to capture formally. Many existing program logics for probabilistic programs either cannot capture these notions or provide poor support.

We begin by describing the law and independence logic IL, a proof system with intuitive rules that are easy to apply and amenable to automation. For simplicity, we only consider programs which sample from the binomial distribution, and have deterministic control flow—for lack of space, we also omit procedure calls.

**Definition 15** (Assertions)**.** *IL assertions have the grammar:*

$$\xi \quad := \quad \mathsf{det}(e) \mid \#E \mid e \sim \mathrm{B}(e, p) \mid \top \mid \bot \mid \xi \wedge \xi$$

*where $e \in \mathcal{E}$, $E \subseteq \mathcal{E}$, and $p \in [0, 1]$.*

The assertion $\mathsf{det}(e)$ states that $e$ is deterministic in the current distribution, i.e., there is at most one element in the support of its interpretation. The assertion $\#E$ states that the expressions in $E$ are independent, as formalized in the previous section. The assertion $e \sim \mathrm{B}(m, p)$ states that $e$ is distributed according to a binomial distribution with parameter $m$ (where $m$ can be an expression) and constant probability $p$, i.e. the probability that $e = k$ is equal to the probability that exactly $k$ independent coin flips return heads using a biased coin that returns heads with probability $p$.

Assertions can be seen as an instance of a logical abstract domain, where the order between assertions is given by implication based on a small number of axioms. Examples of such axioms include independence of singletons, irreflexivity of independence, anti-monotonicity of independence, an axiom for the sum of binomial distributions, and rules for deterministic expressions:

$$\#\{x\} \qquad \#\{x, x\} \iff \mathsf{det}(x) \qquad \#(E \cup E') \implies \#E$$

$$e \sim \mathrm{B}(m, p) \wedge e' \sim \mathrm{B}(m', p) \wedge \#\{e, e'\} \implies e + e' \sim \mathrm{B}(m + m', p)$$

$$\bigwedge_{1 \leq i \leq n} \mathsf{det}(e_i) \implies \mathsf{det}(f(e_1, \ldots, e_n))$$

**Definition 16.** *Judgments of the logic are of the form $\{\xi\}\ s\ \{\xi'\}$, where $\xi$ and $\xi'$ are IL-assertions. A judgment is* valid *if it is derivable from the rules of Fig. 9 (structural rules and rule for sequential composition are similar to those from § 4 and omitted.*

The rule [IL-ASSGN] for deterministic assignments is as in § 4. The rule [IL-SAMPLE] for random assignments yields as post-condition that the variable $x$ and a set of expressions $E$ are independent assuming that $E$ is independent before the sampling, and moreover that $x$ follows the law of the distribution that it is sampled from. The rule [IL-COND] for conditionals requires that the guard is deterministic, and that each of the branches satisfies the specification; if the guard is not deterministic, there are simple examples where the rule is not sound.The rule [IL-WHILE] for loops requires that the loop is certainly terminating with a deterministic guard. Note that the requirement of certain termination could be avoided by restricting the structural rules such that a statement $s$ has deterministic control flow whenever $\{\xi\}\ s\ \{\xi'\}$ is derivable.

We now turn to the embedding. The embedding of IL assertions into general assertions is immediate, except for $\mathsf{det}(e)$ which is translated as $\square e \vee \square \neg e$. We let $\overline{\xi}$ denote the translation of $\xi$.

**Theorem 17** (Embedding and soundness of IL logic)**.** *If $\{\xi\}\ s\ \{\xi'\}$ is derivable with the rules of the IL logic, then $\{\overline{\xi}\}\ s\ \{\overline{\xi'}\}$ is derivable in (the syntactic variant of) Ellora. As a consequence, every derivable judgment $\{\xi\}\ s\ \{\xi'\}$ is valid.*

*Proof sketch.* By induction on the derivation. The interesting cases are conditionals and loops. For conditionals, the soundness follows from the soundness of the rule in the general proof system:

$$\frac{\{\eta\}\ s_1\ \{\eta'\} \qquad \{\eta\}\ s_2\ \{\eta'\} \qquad \square e \vee \square \neg e}{\{\eta\}\ \textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2\ \{\eta'\}}$$

$$\text{IL-ASSGN}\ \frac{}{\{\xi[x := e]\}\ x \leftarrow e\ \{\xi\}}$$

$$\text{IL-SAMPLE}\ \frac{\{x\} \cap \mathrm{FV}(E) \cap \mathrm{FV}(e) = \emptyset}{\{\#E\}\ x \xleftarrow{\$} \mathrm{B}(e,p)\ \{\#(E \cup \{x\}) \wedge x \sim \mathrm{B}(e,p)\}}$$

$$\text{IL-COND}\ \frac{\{\xi\}\ s_1\ \{\xi'\} \qquad \{\xi\}\ s_2\ \{\xi'\} \qquad \xi \implies \mathsf{det}(b)}{\{\xi\}\ \mathbf{if}\ b\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2\ \{\xi'\}}$$

$$\text{IL-WHILE}\ \frac{\{\xi\}\ s\ \{\xi\} \qquad \xi \implies \mathsf{det}(b) \qquad \mathcal{C}_{\text{CTerm}}}{\{\xi\}\ \mathbf{while}\ b\ \mathbf{do}\ s\ \{\xi\}}$$

**Figure 9.** Selected proof rules of IL logic

To prove the soundness of this rule, we proceed by case analysis on $\Box e \vee \Box\neg e$. We treat the case $\Box e$; the other case is similar. In this case, $\eta$ is equivalent to $\eta_1 \wedge \Box e \oplus \eta_2 \wedge \Box\neg e$, where $\eta_1 = \eta$ and $\eta_2 = \bot$. Let $\eta_1' = \eta'$ and $\eta_2 = \Box\bot$; again, $\eta_1' \oplus \eta_2'$ is logically equivalent to $\eta'$. The soundness of the rule thus follows from the soundness of the [COND] and [CONSEQ] rules.

For loops, there exists a natural number $n$ such that $\mathbf{while}\, b\, \mathbf{do}\, s$ is semantically equivalent to $(\mathbf{if}\ b\ \mathbf{then}\ s)^n$. By assumption $\{\xi\}\ s\ \{\xi\}$ holds, and thus by induction hypothesis $\{\bar{\xi}\}\ s\ \{\bar{\xi}\}$. We also have $\xi \implies \mathsf{det}(b)$, and hence $\{\bar{\xi}\}\ \mathbf{if}\ b\ \mathbf{then}\ s\ \{\bar{\xi}\}$. We conclude by using the [SEQ] rule. □

```
proc sum () =
  var s:int, x:int;
  s ← 0;
  for j ← 1 to N do
    x ←$ B(j,1/2);
    s ← s + x;
  return s
```

**Figure 10.** Sum of bin.

To illustrate our system IL, consider the statement $s$ in Fig. 10 which flips a fair coin $N$ times and counts the number of heads. Using the logic, we can prove that $\mathsf{s} \sim \mathrm{B}(N \cdot (N+1)/2, 1/2)$ is a valid post-condition for $s$. We omit the proof that the loop guard is deterministic, and focus on the distribution of $s$. We take the following invariant:

$$\mathsf{s} \sim \mathrm{B}\left(\mathsf{j}(\mathsf{j}+1)/2, 1/2\right)$$

The invariant holds initially, as $0 \sim B(0, 1/2)$. For the inductive case, we have to establish

$$\{\mathsf{s} \sim \mathrm{B}\left(0, 1/2\right)\}\ c_0\ \{\mathsf{s} \sim \mathrm{B}\left((\mathsf{j}+1)(\mathsf{j}+2)/2, 1/2\right)\}$$

where $c_0$ represents the loop body, i.e. $\mathsf{x} \xleftarrow{\$} \mathrm{B}\left(\mathsf{j}, 1/2\right); \mathsf{s} \leftarrow \mathsf{s} + \mathsf{x}$. First, we apply the rule for sequence taking as intermediate assertion

$$s \sim \mathrm{B}\left(\mathsf{j}(\mathsf{j}+1)/2, 1/2\right) \wedge \mathsf{x} \sim \mathrm{B}\left(\mathsf{j}, 1/2\right) \wedge \#\{\mathsf{x}, \mathsf{s}\}$$

The first premise follows from the rule for random assignment and structural rules. The second premise follows from the rule for deterministic assignment and the rule of consequence, applying axioms about sums of binomial distributions.

We briefly comment on several limitations of IL. First, IL is restricted to programs with deterministic control flow, but this restriction could be partially relaxed by enriching IL with assertions for conditional independence. Such assertions are already expressible in the logic of Ellora; adding conditional independence would significantly broaden the scope of the IL proof system and open the possibility to rely on axiomatizations of conditional independence (e.g., based on graphoids [40]). Second, the logic only supports sampling from binomial distributions. It is possible to enrich the language of assertions with clauses $s \sim g$ where $g$ can model other distributions, like the uniform distribution or the Laplace distribution. The main design challenge is finding a core set of useful facts about these distributions. Enriching the logic and automating the analysis are interesting avenues for further work.

## 7. Case studies

In this section, we will demonstrate Ellora on a selection of examples.[3] Together, they exhibit a wide variety of different proof techniques and reasoning principles, while demonstrating various uses of randomization in algorithm design. Our examples freely use facilities for defining custom operations and lemmas in the assertion logic, which are available in Ellora's implementation.

***Hypercube routing.*** We will begin with the *hypercube routing* algorithm [47, 48]. To set the stage, consider a network where each node is labeled by a bitstring of length $D$, and two nodes are connected by an edge if and only if the two corresponding labels differ in exactly one bit position. This network topology is known as a *hypercube*, a $D$-dimensional version of the standard cube; a simple example with $D = 3$ is in Fig. 11. In the network, there is initially one packet at each node, and each packet has a unique destination. Our goal is to design a routing strategy that will move the packets from node to node, following the edges, until all packets reach their destination. Furthermore, the routing should be *oblivious*: to avoid coordination overhead, each packet must select a path without considering the behavior of the other packets.

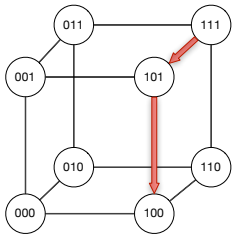


**Figure 11.** Hypercube path from 111 to 100 ($D = 3$)

To model the flow of packets, each packet's current position is a node in the graph. Time proceeds in a series of steps, and at each step at most one packet can traverse any single edge. If several packets try to use the same edge simultaneously, one packet will be selected to move (for any selection strategy that selects *some* packet). The other packets wait at their current position; these packets are *delayed* and make no progress this step.

The routing strategy is based on *bit fixing*: if the current position has bitstring $i$, and the target node has bitstring $j$, we compare the bits in $i$ and $j$ from left to right, moving along the edge that corrects the first differing bit. For instance, if we are at 111 and we wish to reach 100, we will move along the edge corresponding to the second position: from 111 to 101, and then along the edge corresponding to the third position: from 101 to 100. See Fig. 11 for a picture.

While this strategy is simple and oblivious, there are permutations $\pi$ that require a total number of steps growing linearly in the number of packets to route all packets. Valiant proposes a simple modification, so that the total number of steps grows *logarithmically* in the number of packets. In the first phase, each packet $i$ will first select an intermediate destination $\rho(i)$ uniformly at random from all nodes, and use bit fixing to reach $\rho(i)$. In the second phase, each packet will use bit fixing to go from $\rho(i)$ to the destination $\pi(i)$. We will focus on the first phase, since the reasoning for the second phase is nearly identical. We can model the strategy with the following code, using some syntactic sugar for the **for** loops. We recall that the number of node in a hypercube of dimension $D$ is $2^D$ so each node can be identified by a number in $[1, 2^D]$.

```
proc route (D T : int) :
  var ρ, pos, usedBy : node map;
```

[3] We present further examples in the supplemental material.

```
var nextE : edge;
pos ← Map.init id 2^D; ρ ←Map.empty;
for i ←1 to 2^D do ρ[i] ←$ [1, 2^D]
for t ←1 to T do
  usedBy ← Map.empty;
  for i ←1 to 2^D do
    if pos[i] ≠ ρ[i] then
      nextE ← getEdge pos[i] ρ[i];
      if usedBy[nextE] = ⊥ then
        usedBy[nextE] ← i; // Mark edge used
        pos[i] ←dest nextE // Move packet
return (pos, ρ)
```

We assume that initially the position of the packet $i$ is at node $i$ (see `Map.init`). Then, we initialize the random intermediate destinations $\rho$. The remaining loop encodes the evaluation of the routing strategy iterated $T$ time. The variable `usedBy` is a map that logs if an edge is already used by a packet, it is empty at the beginning of each iteration. For each packet, we try to move it across one edge along the path to its intermediate destination. The function `getEdge` returns the next edge to follow, following the bit-fixing scheme. If the packet can progress (its edge is not used), then its current position is updated and the edge is marked as used.

Our goal is to show that if the number of timesteps $T$ is $4D + 1$, then all packets reach their intermediate destination in at most $T$ steps, except with a small probability $2^{-2D}$ of failure. That is, the number of timesteps grows linearly in $D$, logarithmic in the number of packets. At a high level, the analysis involves three steps.

The first step is to consider a single packet traveling from $i$ to $\rho(i)$, and to calculate the average number of other packets that share at least one edge with $i$'s path $P$. Let $H^\rho(i, j)$ be 1 if the paths of packets $i$ and $j$ share at least one edge and 0 otherwise. Then, the load $R^\rho(i)$ on $i$'s path is the sum of $H^\rho(i, j)$ over all the other packets $j$. By using the fact that each intermediate destination in $\rho$ is uniformly distributed, and by analyzing the number of packets beside $i$'s that use each edge on $i$'s path, we can upper bound the expected value of $R^\rho(i)$ by $D/2$. In the second step, we move from a bound on the expectation of $R^\rho(i)$ to a *high-probability bound*: we want to show that $R\rho(i) < 3D$ holds except with a small probability of failure. The key tool is the *Chernoff bound*, which gives high probability bounds of sums of independent samples. The libraries in Ellora include a mechanized proof of a generalized Chernoff bound, with the following statement (leaving off some of the parameters):

```
lemma Chernoff n(d:mem distr)(X:int → mem → bool):
  let Xm = Σ_{i∈[1,n]} X i m in
  E[X|d] ≤ μ ∧ indep [0, n] (X i) d
  ⇒ Pr[X > (1 + δ)μ|d] ≤ (e^δ/(1 + δ)^{1+δ})^μ
```

Returning to the proof, we bounded the expectation of $R$ in the previous step. To apply the Chernoff bound, we need to show that for any packet $i$, the expressions $H^\rho(i, j)$ are independent for all $j$. This is not exactly true, since $H^\rho(i, j_1)$ and $H^\rho(i, j_2)$ both depend on the (random) destination $\rho(i)$ of packet $i$. However, it suffices to show that these variables are independent if we fix the value of $\rho(i)$; then we can apply the Chernoff bound to upper bound $R$ with high probability.

Finally, we can bound the total delay of any packet. This portion of the proof rests on an intricate loop invariant assigning an imaginary coin for each delay step to some packet that crosses $P$. By showing that each packet holds at most one coin, we can conclude that the $i$'s delay is at most the number $R(i)$ of crossing packets. With our high probability bound from the previous step, we can show that $T = 4D + 1$ timesteps is sufficient to route all packets $i$ to $\rho(i)$, except with some small probability:

$$\{T = 4D + 1\}\ \texttt{route}\ \{\Pr[\exists i.\ \texttt{pos}[i] \neq \rho[i]] \leq 2^{-2D}]\}$$

***Modeling infinite processes.*** Our second example is the *coupon collector* process. The algorithm draws a uniformly random coupon (we have $N$ coupon) on each day, terminating when it has drawn at least one of each kind of coupon. The code of the algorithm is displayed in Figure 12. The code uses the array `cp` to keep track of the coupons seen so far; initially $\texttt{cp}[i] = 0$. We divide the loop into a sequence of phases (the outer loop) where in each phase we repeatedly sample coupons and wait until we see a new coupon (the inner loop). We keep track of the number of steps we spend in each phase $p$ in $\texttt{t}[p]$, and the total number of steps in $X$. Our goal is to bound the average number of iterations. The code involves two nested loops, so we have two loop invariants. The inner loop has a probabilistic guard: at every iteration, there is a finite probability that the loop terminates (i.e., if we draw a new coupon), but there is no finite bound on the number of iterations we need to run. Since our desired loop invariant is not downwards closed, we must apply the rule for almost sure termination.

```
proc coupon (N : int) :
  var int cp[N], t[N];
  var int X ←0;
  for p ←1 to N do
    ct ←0;
    cur ←$ [1, N];
    while cp[cur] = 1 do
      ct ←ct + 1;
      cur ←$ [1, N];
    t[p] ←ct;
    cp[cur] ← 1;
    X ← X + t[p];
  return X
```

**Figure 12.** Coupon collector

We use a variant that is 1 if we have not seen a new coupon, and 0 if we have seen a new coupon. Note that each iteration, we have a strictly positive probability $\rho(p)$ of seeing a new coupon and decreasing the variant. Furthermore, the variant is bounded by 1, and the loop exits when the variant reaches 0. So, the side condition $\mathcal{C}_{\text{ASTerm}}$ holds. For the inner loop, we prove that forall $c$ the invariant $\eta_i$ is preserved:

$$\begin{cases} (\Box(\texttt{cp[cur]} = 1 \Rightarrow c \leq \texttt{ct}) \\ \wedge \Pr[\texttt{cp[cur]} = 0 \wedge c = \texttt{ct}] = (1 - \rho(p))^c \rho(p)) \\ \vee \\ (\exists k \in [1, c).\ \Box(\texttt{cp[cur]} = 1 \Rightarrow \texttt{ct} = k) \\ \wedge \Box(\texttt{cp[cur]} = 0 \Rightarrow \texttt{ct} \leq k) \\ \wedge \Pr[\texttt{cp[cur]} = 1 \wedge \texttt{ct} = k] = (1 - \rho(p))^k). \end{cases}$$

Note that this is a $t$-closed formula; there is an existential in the second disjunction, but it has finite domain (for fixed $c$).

For intuition, for every natural number $c$ there are two cases: Either we have already unrolled more than $c$ iterations, or not. The first disjunction corresponds to the first case, since loops where the guard is true all have $\texttt{ct} \geq c$, and the probability of stopping at $c$ iterations is $(1 - \rho(p))^c \rho(p)$—we see $c$ old coupons, and then a new one. Otherwise, we have the second disjunction. The integer $k$ represents the current number of unfoldings of the loop. If the loop is continuing then $k = \texttt{ct}$. If the loop is terminated, it terminated before the current iteration: $\texttt{ct} < k$. Furthermore, the probability of continuing at iteration $k$ is $(1 - \rho(p))^k$. At the end of the loop we have $\Box\texttt{cp[cur]} = 0$. So, by the first conjunct and some manipulations,

$$\forall c \in \mathbb{N}.\ \Pr[c = \texttt{ct}] = (1 - \rho(p))^c \rho(p)$$

holds when the inner loop exits, precisely describing the distribution of iterations `ct` as $\mathbf{Geom}(\rho(p))$ by definition.

The outer loop is easier to handle, since the loop has a fixed bound $N$ on the number of iterations so we can use the rule for

certain termination. For the loop invariant, we take:

$$\eta_{out} \triangleq \begin{cases} \forall i \in [p-1].\ t[i] \sim \mathtt{Geom}(\rho(i)) \wedge \Box\Big(X = \sum_{i\in[p-1]} t[i]\Big) \\ \wedge\ \Box\Big(\sum_{i\in[1,N]} \mathtt{cp}[i] = p-1\Big) \\ \wedge\ \forall i \in [1,N].\ \Box(\mathtt{cp}[i] \in \{0,1\}). \end{cases}$$

The first conjunct states that the previous waiting times follow a geometric distribution with parameter $\rho(i)$; this assertion follows from the previous reasoning on the inner loop. The second conjunct asserts that $X$ holds the total waiting time so far. The final two conjuncts state that there are at most $p-1$ flags set in cp. Thus,

$$\{\mathcal{L}\}\ \mathtt{coupon}\ \left\{ \begin{array}{l} \forall i \in [1,N].\ t[i] \sim \mathtt{Geom}(\rho(i)) \\ \wedge\ \Box X = \sum_{i\in[1,N]} t[i] \end{array} \right\}$$

at the end of the outer loop. By applying linearity of expectations and a fact about the expectation of the geometric distribution, we can bound the expected running time:

$$\{\mathcal{L}\}\ \mathtt{coupon}\ \left\{\mathbb{E}[X] = \textstyle\sum_{i\in[1,N]} \left(\frac{N}{N-i+1}\right)\right\}.$$

*Comparison with Kaminski et al. [27].* Coupon collector provides a point of comparison with expectation-based techniques, as Kaminski et al. [27] have independently proved the example using a system inspired by pGCL. At a high level, their analysis involves complex equations, in place where we require rather complex invariants. The two approaches are difficult to compare, in the sense that it seems hard to infer their equations from our invariants and vice-versa. However, we believe that the logical invariants used by our proof are more intuitive and easier to discover than the equations used in the pGCLproof. Moreover, we note that Ellora naturally supports reasoning about independence, and could be used to reason about the variance of $X$, whereas reasoning about independence in pGCL remains possible but challenging.

```
proc pwInd (N : int) :
  var bool X[2^N], B[N];
  for i ← 1 to N do
    B[i] ←$ Ber(1/2);
  for j ← 1 to 2^N do
    X[j] ← 0;
    for k ← 1 to N do
      if k ∈ bits(j) then
        X[j] ← X[j] ⊕ B[k]
  return X
```

**Figure 13.** Pairwise Indep.

***Limited randomness.*** As we have seen in the routing example, independent random bits are a powerful tool in randomized algorithms. However, they are also scarce resource—fresh randomness for each bit is needed. For many applications, e.g. hashing, the weaker notion of *pairwise independence* suffices. Informally, pairwise independence says that if we see the result of $X_i$, we do not gain information about all other variables $X_k$. However, if we see the result of *two* variables $X_i, X_j$, we may gain information about $X_k$.

There are many constructions in the algorithms literature that grow a small number of independent bits into more pairwise independent bits. Figure 13 gives one procedure, where $\oplus$ is exclusive-or, and bits(j) is the set of positions set to 1 in the binary expansion of j. The proof uses the following fact, which we fully verify: for a uniformly distributed Boolean random variable $Y$, and a random variable $Z$ of any type,

$$Y \,\#\, Z \Rightarrow Y \oplus f(Z) \,\#\, g(Z) \tag{1}$$

for any two Boolean functions $f, g$. Then, note that

$$\mathtt{X}[i] = \bigoplus_{\{j\in\mathtt{bits}(i)\}} \mathtt{B}[j]$$

```
var H: ({0,1}^l, {0,1}^l) map;
proc orcl (q:{0,1}^l):
  var a : {0,1}^l;
  if q ∉ H then
    a ←$ {0,1}^l;
    bad ← bad || a ∈ codom(H);
    H[q] ← a;
  return H[q];

proc main():
  var b: bool;
  bad ← false;
  H ← [];
  b ← A();
  return b;
```

**Figure 14.** PRP/PRF game

where the big XOR operator ranges over the indices $j$ where the bit representation of $i$ has bit $j$ set. For any two $i, k \in [1, \ldots, 2^{\mathtt{N}}]$ distinct, there is a bit position in $[1, \ldots, \mathtt{N}]$ where $i$ and $k$ differ; call this position $r$ and suppose it is set in $i$ but not in $k$. By rewriting,

$$\mathtt{X}[i] = \mathtt{B}[r] \oplus \bigoplus_{\{j\in\mathtt{bits}(i)\setminus r\}} \mathtt{B}[j] \quad \text{and} \quad \mathtt{X}[k] = \bigoplus_{\{j\in\mathtt{bits}(k)\setminus r\}} \mathtt{B}[j].$$

Since $\mathtt{B}[j]$ are all independent, $\mathtt{X}[i]$ # $\mathtt{X}[k]$ follows from Eq. (1) taking $Z$ to be the distribution on tuples $\langle \mathtt{B}[1], \ldots, \mathtt{B}[\mathtt{N}]\rangle$ excluding $\mathtt{B}[r]$. This verifies pairwise independence:

$$\{\mathcal{L}\}\ \mathtt{pwInd(N)}\ \{\mathcal{L} \wedge \forall i, k \in [2^{\mathtt{N}}].\ i \neq k \Rightarrow \mathtt{X}[i] \,\#\, \mathtt{X}[k]\}.$$

***Adversarial programs.*** Pseudorandom functions (PRF) and pseudorandom permutations (PRP) are two idealized primitives that play a central role in the design of symmetric-key systems. Although the most natural assumption to make about a blockcipher is that it behaves as a pseudorandom permutation, most commonly the security of such a system is analyzed by replacing the blockcipher with a perfectly random function. The PRP/PRF Switching Lemma [8, 25] fills the gap: given a bound for the security of a blockcipher as a pseudorandom function, it gives a bound for its security as a pseudorandom permutation.

**Lemma 18** (PRP/PRF switching lemma)**.** *Let $A$ be an adversary with blackbox access to an oracle $O$ implementing either a random permutation on $\{0,1\}^l$ or a random function from $\{0,1\}^l$ to $\{0,1\}^l$. Then the probability that the adversary $A$ distinguishes between the two oracles in less that $q$ calls is bounded by $\frac{q(q-1)}{2^{l+1}}$:*

$$|\Pr_{PRP}[b \wedge |H| \leq q] - \Pr_{PRF}[b \wedge |H| \leq q]| \leq \frac{q(q-1)}{2^{l+1}}$$

*where $H$ is a map storing each call performed by the adversary and $|H|$ the size of $H$.*

Proving this lemma can be done using the Fundamental Lemma of Game-Playing, and bounding the probability of *bad* in the program from Fig. 14. We focus on the latter. Here we apply the [ADV] rule of Ellora with the following invariant:

$$\forall k, \Pr[\mathsf{bad} \wedge |H| \leq k] \leq \frac{k(k-1)}{2^{l+1}}$$

where $|H|$ is the size of the map $H$, i.e. the number of adversary call. Intuitively, the invariant says that at each call to the oracle the probability that bad has been set before and that the number of adversary call is less than $k$ is bounded by a polynomial in $k$.

The invariant is $d$-closed and true before the adversary call, since at that point $\Pr[\mathsf{bad}] = 0$. Then we need to prove that the oracle preserves the invariant, which can be done easily using the precondition calculus ([PC] rule).

## 8. Implementation and mechanization

We have built a prototype implementation of Ellora within EasyCrypt [4, 5], a tool-assisted framework originally designed for ver-

ifying proofs of cryptographic protocols. EasyCrypt provides a convenient environment for constructing proofs in various Hoare logics, supporting interactive, tactic-based proofs for manipulating assertions and allowing users to invoke external tools, like SMT-solvers, to discharge simpler proof obligations. Moreover, EasyCrypt provides a mature set of libraries for both data structures (sets, maps, lists, arrays, etc.) and formalized mathematical theorems (algebra, real analysis, etc.), which we extended with theorems from probability theory. We used the implementation for verifying many examples from the literature, including all the programs presented in § 7 as well as some additional examples (such as polynomial identity test, private running sums, properties about random walks, etc.). The verified proofs bear a strong resemblance to the existing, paper proofs.

Independently of this work, Ellora has been used to formalize the main theorem about a randomized gossip-based protocol for distributed systems [30, Theorem 2.1], and there are ongoing efforts to formalize algorithms from blockchain systems like Bitcoin.

## 9. Related work

There is a long tradition of research on semantics and verification of probabilistic programs.

*Semantics of programming languages* Kozen [31] was the first to consider a semantics for probabilistic programs, introducing ordered Banach spaces as a key mathematical tool for interpreting programs and giving a sub-distribution transformer semantics for commands; our work uses a discrete version of his semantics. Alternatively, probabilistic programs can be given a semantics by viewing distributions as monadic values [26, 41]. Semantics of probabilistic programs remains an active topic of research, with recent developments such as [33].

*Expectation-based techniques.* Expectation-based techniques are one of the most well-studied ideas for deductive verification of probabilistic programs. These approaches are inspired by standard techniques for verifying deterministic programs, with expectations playing the role of propositions.

One of the first such formalisms was *Probabilistic Propositional Dynamical Logic* (PPDL), proposed by Kozen [32], drawing a close analogy to Propositional Dynamical Logic. The central objects in this logic are real-valued functions $f, g$ on program states, along with rules for constructing and reasoning about the expected value of $f$ and $g$ on the output distribution of a program $c$. The language of PPDL is quite general—there are commands for filtering and unbounded iteration $c^*$, which can be used to define the usual commands in an imperative language.

In a series of works initiated by Morgan et al. [39] and described in their textbook [35], McIver, Morgan, and their collaborators extended ideas from PPDL to study an imperative language with probabilistic choice and non-determinism called *probabilistic Guarded Command Language* (pGCL). Like PPDL, pGCL reasons about the expected value of a single real-valued function on program states. The central tool of pGCL is *greatest pre-expectation*, a mechanical procedure similar to Dijkstra's weakest-pre-condition but transforming expectations instead of predicates. Many subsequent works build on pGCL [17, 18, 24, 28] or use related ideas [1, 27]. In particular, Kaminski et al. [27] give a calculus for bounding expected running time, with support for probabilistic loops that terminate almost surely. They also analyze the coupon collector example of § 7.

*Program logics for probabilistic programs.* Instead of reasoning about a single probability or expected value, a different line of research investigates Hoare logics for probabilistic programs, where the pre-condition and post-condition are probabilistic assertions about the input and output distributions. The earliest system is due to Ramshaw [42], who proposes a program logic where assertions can be formulas involving *frequencies*, essentially probabilities on sub-distributions. Ramshaw's logic allows assertions to be combined with operators like $\oplus$, similar to our approach. More recently, den Hartog [15] presents a Hoare-style logic with supporting general assertions on the distribution, allowing expected values and probabilities. However his **while** rule is based on a semantic condition on the guarded loop body, which is less desirable for verification because it requires reasoning about the semantics of programs. Chadha et al. [9] give decidability results for a probabilistic Hoare logic without **while** loops. We are not aware of any existing system that supports assertions about general expected values; existing works also restrict to Boolean distributions. Rand and Zdancewic [43] formalize a Hoare logic for probabilistic programs but unlike our work, their assertions are interpreted on *distributions* rather than sub-distributions. For conditionals, their semantics rescales the distribution of states that enter each branch. However, their assertion language is restricted and they impose strong restrictions on loops.

*Formalizations of probability theory.* Formalizations of measure and integration theory in general purpose interactive theorem provers have been considered in many works [1, 13, 21, 22, 36, 44]. Avigad et al. [2] recently completed a proof of the Central Limit theorem, which is the principle underlying concentration bounds. Hölzl [20] formalized discrete-time Markov chains and Markov Decision Processes. These, and other, existing formalizations have been used to verify several case studies, but they are scattered and not easily accessible for our purposes.

*Other approaches.* There have been many other significant works to verify probabilistic program using different formal approaches. For instance, verification of Markov transition systems goes back to at least Hart et al. [19], Sharir et al. [46]; our condition for ensuring almost-sure termination in loops is directly inspired by their work. Automated methods include model checking (see e.g., [3, 29, 34]) and abstract interpretation (see e.g., [14, 37]). For analyzing probabilistic loops in particular, there are tools for reasoning about running time. There are also automated systems for synthesizing invariants [6, 12]. Chakarov and Sankaranarayanan [10, 11] use a martingale method to compute the expected time of the coupon collector process for $N = 5$—fixing $N$ lets them focus on a program where the outer **while** loop is fully unrolled. Martingales are also used by Fioriti and Hermanns [16] for analyzing probabilistic termination. Finally, there are approaches involving symbolic execution; Sampson et al. [45] use a mix of static and dynamic analysis to check probabilistic programs from the approximate computing literature.

## 10. Conclusion and perspective

Ellora is a general-purpose framework for verification of randomized programs. We have proved its soundness, and its expressiveness through representative examples from the literature. Prime targets for future formalization include accuracy of differentially private algorithms, lower bounds, distributed algorithms, and amortized complexity. We also hope to apply Ellora to more mathematical areas, like combinatorics proofs based on the probabilistic method. Finally, we plan to extend and automate the IL logic.

## References


[1] P. Audebaud and C. Paulin-Mohring. Proofs of randomized algorithms in Coq. *Sci. Comput. Program.*, 74(8):568–589, 2009.

[2] J. Avigad, J. Hölzl, and L. Serafin. A formally verified proof of the Central Limit Theorem. *CoRR*, abs/1405.7012, 2014.

[3] C. Baier. Probabilistic model checking. In J. Esparza, O. Grumberg, and S. Sickert, editors, *Dependable Software Systems Engineering*, volume 45 of *NATO Science for Peace and Security Series - D:*

*Information and Communication Security*, pages 1–23. IOS Press, 2016.

[4] G. Barthe, B. Grégoire, S. Heraud, and S. Z. Béguelin. Computer-aided security proofs for the working cryptographer. In P. Rogaway, editor, *Advances in Cryptology - CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 71–90. Springer, 2011.

[5] G. Barthe, F. Dupressoir, B. Grégoire, C. Kunz, B. Schmidt, and P. Strub. Easycrypt: A tutorial. In A. Aldini, J. Lopez, and F. Martinelli, editors, *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*, volume 8604 of *Lecture Notes in Computer Science*, pages 146–166. Springer, 2013.

[6] G. Barthe, T. Espitau, L. M. Ferrer Fioriti, and J. Hsu. Synthesizing probabilistic invariants via Doob's decomposition. In *International Conference on Computer Aided Verification (CAV), Toronto, Ontario*, 2016. To appear.

[7] G. Barthe, M. Gaboardi, B. Grégoire, J. Hsu, and P.-Y. Strub. A program logic for union bounds. In *International Colloquium on Automata, Languages and Programming (ICALP), Rome, Italy*, 2016. To appear.

[8] M. Bellare and P. Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In S. Vaudenay, editor, *Advances in Cryptology - EUROCRYPT 2006, 25th Annual International Conference on the Theory and Applications of Cryptographic Techniques, St. Petersburg, Russia, May 28 - June 1, 2006, Proceedings*, volume 4004 of *Lecture Notes in Computer Science*, pages 409–426. Springer, 2006. ISBN 3-540-34546-9.

[9] R. Chadha, L. Cruz-Filipe, P. Mateus, and A. Sernadas. Reasoning about probabilistic sequential programs. *Theoretical Computer Science*, 379(1-2):142–165, 2007.

[10] A. Chakarov and S. Sankaranarayanan. Probabilistic program analysis with martingales. In *International Conference on Computer Aided Verification (CAV), Saint Petersburg, Russia*, volume 8044 of *Lecture Notes in Computer Science*, pages 511–526. Springer, 2013.

[11] A. Chakarov and S. Sankaranarayanan. Expectation invariants as fixed points of probabilistic programs. In *Static Analysis Symposium (SAS)*, volume 8723 of *Lecture Notes in Computer Science*, pages 85–100. Springer-Verlag, 2014.

[12] K. Chatterjee, H. Fu, P. Novotný, and R. Hasheminezhad. Algorithmic analysis of qualitative and quantitative termination problems for affine probabilistic programs. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), St Petersburg, Florida*, pages 327–342, 2016.

[13] A. R. Coble. Anonymity, information, and machine-assisted proof. Technical Report UCAM-CL-TR-785, University of Cambridge, Computer Laboratory, 2010.

[14] P. Cousot and M. Monerau. Probabilistic abstract interpretation. In H. Seidl, editor, *21st European Symposium on Programming, ESOP 2012*, volume 7211 of *Lecture Notes in Computer Science*, pages 169–193. Springer, 2012.

[15] J. den Hartog. *Probabilistic extensions of semantical models*. PhD thesis, Vrije Universiteit Amsterdam, 2002.

[16] L. M. F. Fioriti and H. Hermanns. Probabilistic termination: Soundness, completeness, and compositionality. In S. K. Rajamani and D. Walker, editors, *Proceedings of the 42nd ACM Symposium on Principles of Programming Languages, POPL 2015*, pages 489–501. ACM, 2015.

[17] F. Gretz, J. Katoen, and A. McIver. Prinsys - on a quest for probabilistic loop invariants. In *Quantitative Evaluation of Systems - 10th International Conference, QEST 2013*, pages 193–208, 2013.

[18] F. Gretz, N. Jansen, B. L. Kaminski, J. Katoen, A. McIver, and F. Olmedo. Conditioning in probabilistic programming. In *Mathematical Foundations of Programming Semantics*, 2015.

[19] S. Hart, M. Sharir, and A. Pnueli. Termination of probabilistic concurrent programs. *ACM Trans. Program. Lang. Syst.*, 5(3):356–380, 1983.

[20] J. Hölzl. Markov chains and Markov decision processes in Isabelle/HOL. 2016.

[21] J. Hölzl and A. Heller. Three chapters of measure theory in Isabelle/HOL. In M. C. J. D. van Eekelen, H. Geuvers, J. Schmaltz, and F. Wiedijk, editors, *Interactive Theorem Proving, ITP 2011*, volume 6898 of *Lecture Notes in Computer Science*, pages 135–151. Springer, 2011.

[22] J. Hurd. Formal verification of probabilistic algorithms. Technical Report UCAM-CL-TR-566, University of Cambridge, Computer Laboratory, 2003.

[23] J. Hurd. Verification of the miller-rabin probabilistic primality test. *J. Log. Algebr. Program.*, 56(1-2):3–21, 2003.

[24] J. Hurd, A. McIver, and C. Morgan. Probabilistic guarded commands mechanized in HOL. *Theor. Comput. Sci.*, 346(1):96–112, 2005.

[25] R. Impagliazzo and S. Rudich. Limits on the provable consequences of one-way permutations. In D. S. Johnson, editor, *Proceedings of the 21st Annual ACM Symposium on Theory of Computing, May 14-17, 1989, Seattle, Washigton, USA*, pages 44–61. ACM, 1989. ISBN 0-89791-307-8.

[26] C. Jones and G. D. Plotkin. A probabilistic powerdomain of evaluations. In *IEEE Symposium on Logic in Computer Science (LICS), Asilomar, California*, pages 186–195, 1989.

[27] B. Kaminski, J.-P. Katoen, C. Matheja, and F. Olmedo. Weakest Precondition Reasoning for Expected Run-Times of Probabilistic Programs. In *European Symposium on Programming (ESOP), Eindhoven, The Netherlands*, Jan. 2016.

[28] J. Katoen, A. McIver, L. Meinicke, and C. C. Morgan. Linear-invariant generation for probabilistic programs: Automated support for proof-based methods. In *International Symposium on Static Analysis (SAS), Perpignan, France*, pages 390–406, 2010.

[29] J.-P. Katoen. The probabilistic model-checking landscape. In *Proceedings of LICS'16*, 2016.

[30] D. Kempe, A. Dobra, and J. Gehrke. Gossip-based computation of aggregate information. In *Foundations of Computer Science, 2003. Proceedings. 44th Annual IEEE Symposium on*, pages 482–491. IEEE, 2003.

[31] D. Kozen. Semantics of probabilistic programs. In *20th IEEE Symposium on Foundations of Computer Science, FOCS 1979*, pages 101–114. IEEE Computer Society, 1979.

[32] D. Kozen. A probabilistic PDL. *J. Comput. Syst. Sci.*, 30(2):162–178, 1985.

[33] D. Kozen. Kolmogorov extension, martingale convergence, and compositionality of processes. In *Proceedings of LICS'16*, number `http://hdl.handle.net/1813/41517`, 2016.

[34] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

[35] A. McIver and C. Morgan. *Abstraction, Refinement, and Proof for Probabilistic Systems*. Monographs in Computer Science. Springer, 2005.

[36] T. Mhamdi, O. Hasan, and S. Tahar. On the formalization of the Lebesgue integration theory in HOL. In *1st International Conference on Interactive Theorem Proving, ITP 2010*, volume 6172 of *Lecture Notes in Computer Science*, pages 387–402. Springer, 2010.

[37] D. Monniaux. Abstract interpretation of probabilistic semantics. In J. Palsberg, editor, *Static Analysis, 7th International Symposium, SAS 2000*, volume 1824 of *Lecture Notes in Computer Science*, pages 322–339. Springer, 2000.

[38] C. Morgan. Proof rules for probabilistic loops. In *Proceedings of the BCS-FACS 7th Conference on Refinement*, FAC-RW'96, 1996.

[39] C. Morgan, A. McIver, and K. Seidel. Probabilistic predicate transformers. *ACM Trans. Program. Lang. Syst.*, 18(3):325–353, 1996.

[40] J. Pearl and A. Paz. Graphoids: Graph-based logic for reasoning about relevance relations. In *ECAI*, pages 357–363, 1986.

[41] N. Ramsey and A. Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Portland, Oregon*, pages 154–165, 2002.

[42] L. H. Ramshaw. *Formalizing the Analysis of Algorithms*. PhD thesis, Computer Science, 1979.

[43] R. Rand and S. Zdancewic. VPHL: A Verified Partial-Correctness Logic for Probabilistic Programs. In *Mathematical Foundations of Program Semantics (MFPS XXXI)*, 2015.

[44] S. Richter. Formalizing integration theory with an application to probabilistic algorithms. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *Theorem Proving in Higher Order Logics, 17th International Conference, (TPHOL) 2004*, volume 3223 of *Lecture Notes in Computer Science*, pages 271–286. Springer, 2004.

[45] A. Sampson, P. Panchekha, T. Mytkowicz, K. S. McKinley, D. Grossman, and L. Ceze. Expressing and verifying probabilistic assertions. In M. F. P. O'Boyle and K. Pingali, editors, *ACM Conference on Programming Language Design and Implementation, PLDI '14*, page 14. ACM, 2014.

[46] M. Sharir, A. Pnueli, and S. Hart. Verification of probabilistic programs. *SIAM J. Comput.*, 13(2):292–314, 1984.

[47] L. G. Valiant. A scheme for fast parallel communication. *SIAM journal on computing*, 11(2):350–361, 1982.

[48] L. G. Valiant and G. J. Brebner. Universal schemes for parallel communication. In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*, STOC '81, pages 263–277, New York, NY, USA, 1981. ACM.