Formal Certification of Randomized Algorithms

Abstract

Randomized algorithms are a rich and fascinating class of algorithms, with broad applications in computer science and beyond. They also pose a formidable challenge for formal certification: even intuitive properties of simple programs can have elaborate proofs, requiring complex probabilistic invariants and sophisticated theorems from probability theory.

We present Ellora, a deductive verification system designed for general properties of randomized algorithms, like probabilistic independence, high-probability bounds on error, and expected running time. Our system consists of three principal components. First, a rich assertion logic that can express the complex properties needed to verify randomized algorithms. Second, a formalized suite of higher-level tools from probability theory including probabilistic independence and expectation, along with common tools like the Chernoff bound and Markov's inequality. Third, a probabilistic program logic for general reasoning about probabilistic programs. Our logic supports useful reasoning principles for programs exhibiting a variety of termination behavior.

We prove soundness for a core version of Ellora, and we demonstrate our tool by formally verifying a broad collection of examples from the algorithms literature. The examples demonstrate various uses of randomization, diverse properties, and different proof styles.

1. Introduction

Randomized algorithms are one of the richest areas in theoretical computer science. Blessed with the power to draw random samples during computation, simple randomized algorithms can achieve efficiency unmatched by known deterministic algorithms. Moreover, they offer provable guarantees that deterministic algorithms simply cannot achieve, like cryptographic indistinguishability and differential privacy.

At first glance, randomized algorithms seem to be a perfect target for formal certification: interesting properties, rigorous proofs that are difficult to check manually, pseudocode often included—what's not to like? In reality, however, formal certification of randomized algorithms has proven to be a formidable task. Broadly, there are three major challenges.

1. Diverse properties and invariants. Properties for randomized algorithms typically model (i) *correctness*: the algorithm should compute the right probabilistic answer; (ii) *precision*: the algorithm should have low probability of failure; and (iii) *computational efficiency*, expressed probabilistically. However, randomized algorithms have many different applications, and there is a multitude of specific properties for various notions of "right answer", "failure", and "efficiency". Furthermore, their proofs can require first showing ad hoc intermediate invariants.

2. Intricate proofs. Algorithms implemented in just a few lines of code can require pages of arguments to justify their correctness. Not only do these proofs reason about the probabilities of various events, they also use higher-level properties like independence of random variables and assertions about the distribution law of certain samples, enabling sophisticated tools from probability theory. A typical tool is a *concentration bound*, which bounds the deviation of a sum of independent random variables from its average. Applying such a bound requires stating (and proving!) probabilistic independence, and getting a handle on the expected value of the sum.

In contrast, existing verification systems work at a low level of abstraction—typically reasoning only about about raw probabilities rather than distributions. This weakness leads to complex formal proofs and renders many algorithms simply infeasible to verify.

3. Complex control flow. Randomized algorithms are often presented as pseudocode in a plain imperative language with commands for random sampling and loops. The complexity stems from the control flow, which can be probabilistic. This setting poses serious challenges for verification; handling even basic properties like termination may not be cut-and-dried, to say nothing of designing useful reasoning principles.

Our contributions. We present Ellora, a verification platform for concise, high-level formal reasoning about randomized algorithms. The system consists of three components.

1. A rich assertion logic. We base our development on a two-layer assertion language (§ 4), which can concisely express key notions like probability, expectation, and independence. The language also supports *big operators*, which are critical for handling the complex invariants that are common in randomized algorithms.

2. High-level libraries for probability theory. We pair the assertion language with extensive formalized libraries for probability theory, including formalizations of common tools like concentration bounds and Markov's inequality, and theorems about probabilistic independence (§ 5). These libraries enable verification to proceed at a level of abstraction closer to the existing paper proofs, enabling more concise and natural formal proofs.

3. A core program logic and metatheory. We define an expressive program logic for pWhile, a core imperative language with probabilistic sampling, and prove its soundness (§ 6). This language is quite general: both the control flow and termination behavior can be probabilistic.

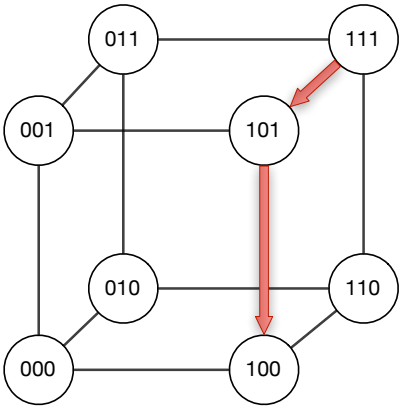

Figure 1. Hypercube path from 111 to 100 ($D = 3$)

To support useful reasoning, we propose two rules for handling **while** loops. Each rule is an instance of the usual rule for deterministic loops, with a pair of additional premises: one to constrain the termination behavior of the loop, the other to constrain the form of the loop invariant for soundness. Our rules under *certain* termination, *almost-sure* termination, and arbitrary termination.

We implement Ellora (§ 7) and demonstrate it on a diverse set of examples from the theoretical computer science literature (§ 8). We are optimistic that machine-checked proofs of complex randomized algorithms are now within reach.

2. A motivating example: hypercube routing

To illustrate the challenges in formally verifying randomized algorithms, we will consider the *hypercube routing* algorithm [28, 29]. While the code is relatively straightforward, the proof is not—this work received the best paper award at the theory conference STOC 1981. Along the way, we will see how our system can handle different parts of this proof.

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 Figure 1). In this 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. In other words, we want to implement a *permutation routing*—we want to permute the packets, where the permutation π is an input to the algorithm. Furthermore, the routing should be *oblivious*: each packet selects a path without considering the behavior of the other packets.

To model the flow of packets, each packet's current position is a node in the graph. We assume that time proceeds in a series of steps, and at each step at most one packet can traverse any single edge. If several packets want to use the same edge simultaneously, one packet will be selected to move (the precise strategy for selecting this packet will not be important, as long as *some* packet is selected). The other packets will wait at their current position; these packets are *delayed* and make no progress this time 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 Figure 1 for a picture.

While this strategy is simple and oblivious, there are permutations π 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 loops).

```
proc route (D T : int) :
var rho, pos, edgeUser : node array;
var path : path; var nextEdge : edge;
rho = pos = Array.make; edgeUser = Array.make;
for i = 0 to (numNodes D) do: rho[i] ←$ unifNode D end
for t = 0 to T do:
  edgeUser = Array.clear edgeUser;
  for i = 0 to (numNodes D) do:
    path = mk i rho[i];
    if (pos[i] ≠ rho[i]) then
      nextEdge = getEdge p pos[i];
      if (edgeUser[e] = ⊥) then
        edgeUser[next] = i; // Mark edge used
        pos[i] = dest e // Move packet
      fi
    fi
  end
end
return (pos,rho)
```

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

Average path load: reasoning about expectation. The first step is to consider a single packet traveling from i to $\rho(i)$, and to calculate the average number $R(i)$ of other packets that share at least one edge with i's path P.

We can specify $R(i)$ in Ellora as follows.

```
op H (i j:node) (rho:node array) =
  1(i ≠ j ∧ cross (mk i rho[i]) (mk j rho[j]))
op R (i:node) (rho:node array) =
  Σ [0, N) (fun j ⇒ H i j rho)
```

Roughly, `H(i, j)` is 1 if the paths of packets i and j share at least one edge, and we define the load `R(i)` on i's path as the sum of `H(i,j)` over all the other packets `j`.

In more detail, Ellora can reason about two layers of expressions and assertions. First, *state expressions* are defined in a single memory, and *state assertions* are predicates on single memories. For instance, the state expression `H` uses the indicator function $\mathbf{1}(\phi)$ which is 0 if the state assertion ϕ is false, 1 if ϕ is true. `H` uses two user-defined operators to build the state assertion: `cross` is true if the two input paths share at least one edge, `mk` takes two nodes a, b and returns the bit-fixing path from a to b. The system also supports *big operators*: Σ is a summation with range and summand given by the two arguments; big operators are needed to sum a variable number of terms.

The second layer of Ellora consists of *probability expressions*, which can represent probabilities and expectations, and *probability assertions*, which are predicates on probability expressions. This layer of Ellora is interpreted in a distribution on memories, rather than a single memory.

The basic probability expression in Ellora is $\oint e$, which stands for the expected value of expression e. Thus, we can define the expectation of the state expression $R(i)$:

$$\mathbb{E}[\texttt{R(i)}] \triangleq \oint \texttt{R(i)}.$$

We can bound this expression by using facts from the theory of expected values and big operators in Ellora. For instance, expectations commute with summations. In mathematics:

$$\mathbb{E}\left[\sum_{i\in[0,N)} X_i\right] = \sum_{i\in[0,N)} \mathbb{E}[X_i].$$

Or as a lemma in Ellora's libraries about expected value:

```
lemma linexp (d:mem distr) (Xi:int ⇒ real var) :
  E[Σ [0, n) Xi | d] =
  Σ [0, n) (fun (j:int) ⇒ E[Xi j | d])
```

Here, `d` is a distribution over program memories `mem`, and `Xi` is a function mapping integers to real-valued variables in the memory, i.e., a family of variables indexed by an integer. The expectation of a variable `X` in distribution `d` is written `E[X | d]`.

High-probability bound: independence and concentration. In the second step, we move from a bound on the expectation of $R(i)$ to a *high-probability bound*: we want to show that $R(i) < 3D$ 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:

```
lemma Chernoff (d:mem distr) (Xi:int → bool var) :
  let X = Σ [0, n) Xi in
  E[X | d] ≤ μ ∧ indep [0, n) Xi d
  ⇒ Pr[X > (1 + δ)μ | d] ≤ (e^δ/(1 + δ)^(1+δ))^μ
```

For simplicity we have left off some of the parameters; see § 5 for more detail. There are two new components. First, the conclusion is a probability assertion bounding the probability. In Ellora, the probability of an event is simply

$$\Pr[\phi] \triangleq \oint \mathbf{1}(\phi),$$

where ϕ is state assertion. The other novelty is the probabilistic assertion

```
indep [0, n) Xi d,
```

which asserts that a set of variables `Xi` for indexes in $[0, n)$ are mutually independent in distribution `d`; this assertion is defined in Ellora's libraries as an assertion on probabilities.

The Chernoff bound is a theorem in pure probability theory, and the proof is complex. The libraries in Ellora include a mechanized proof of a generalized Chernoff bound, as well as other useful theorems like Markov's inequality.

Back to the proof, we bounded the expectation of R in the previous step. To apply the bound, we just need to show that for any packet i, the expressions $H(i, j)$ are independent for all j. This is not precisely true, since $H(i, j_1)$ and $H(i, j_2)$ 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 $\rho(i)$; more precisely, $H(i, j_1)$ and $H(i, j_2)$ are conditionally independent. The proof relies on Ellora's formalization of independence of random variables and related theorems.

Bounding the delay. Finally, we put everything together to bound the total delay of any packet. This portion of the proof utilizes the program logic of Ellora, and 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. More precisely, we can prove the following judgment in Ellora's program logic:

$$\{T = 4D{+}1\}\ \texttt{route}(D, T)\ \{\Pr[\exists i.\ \texttt{pos}[i] \neq \texttt{rho}[i]] \leq \beta(D)]\}$$

where $\beta(D) = 2^{-2D}$, as desired.

This example demonstrates many features of our system but it does not show an important class of reasoning: handling algorithms with *probabilistic loops*, where the guard depends on samples from the loop body. Our system has several features (§ 6) for such programs, involving a careful termination analysis. We will see these features later in the paper.

3. Programs

We base our development on pWhile, an extension of the While language with two constructs: **abort**, which halts

the computation with no result, and probabilistic assignment $x \xleftarrow{\$} g$, which assigns a value sampled according to the distribution g to the program variable x. The syntax of statements is defined by the grammar:

$$s ::= \mathbf{skip} \mid \mathbf{abort} \mid x := e \mid x \xleftarrow{\$} g \mid s; s \\ \mid \mathbf{if}\ e\ \mathbf{then}\ s\ \mathbf{else}\ s \mid \mathbf{while}\ e\ \mathbf{do}\ s$$

where x, e, and g range over 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 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 $\mathbb{D}\ T$ the type for probability distributions over the ground type T. Ellora can be flexibly extended with user-defined functions, symbols, and types.

Semantics. Our semantics is based on *sub-distributions* over a discrete (finite or countably infinite) set A, i.e., a function $\mu : A \to \mathbb{R}^+$ such that

$$\mathrm{wt}(\mu) = \sum_{a \in A} \mu(a) \leq 1.$$

When the weight $\mathrm{wt}(\mu)$ is equal to 1, we call μ a *(proper) distribution*. We let $\mathbf{Distr}\ A$ denote the set of sub-distributions over A. A trivial example of a sub-distribution is the *null sub-distribution* $\mathbf{0}_A \in \mathbf{Distr}\ A$, which maps each element of A to 0. Note that the *probabilities* of μ can be real numbers; in particular, $\mathbf{Distr}\ A$ is not discrete.

We interpret every ground type T as a discrete set $[\![T]\!]$ and $\mathbb{D}$ as the function that maps every discrete set A to the set $\mathbf{Distr}\ A$; other constructors C are interpreted as functions $[\![C]\!]$ from **Set** to **Set** such that the image of a discrete set is discrete. The set State contains well-typed finite maps from variables to values, where values are elements of $\mathcal{V} = \bigcup_T [\![T]\!]$, with T a discrete type. State is countable by construction. The set pState of probabilistic states is $\mathbf{Distr}$ State. One can equip pState with the standard monadic constructions for lifting a state to a probabilistic state $(\mathsf{unit}\ m)$, and for monadic composition $(\mathsf{Mlet}\ x = \mu\ \mathsf{in}\ M)$.

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 proper distributions. The semantics $[\![s]\!]_\mu$ of a statement is parametrized by a probabilistic state μ, and is defined by the equations of Figure 2. It will be convenient to have an explicit semantics of **while** as the limit of its truncated iterates. Given a loop $\mathbf{while}\ b\ \mathbf{do}\ s$, we define:

- its nth *truncated iterate* as $(\mathbf{if}\ b\ \mathbf{then}\ s)^n; \mathbf{assert}\ \neg b$
- its nth *iterate* as $(\mathbf{if}\ b\ \mathbf{then}\ s)^n$,

for every natural number n, where $\mathbf{if}\ b\ \mathbf{then}\ s$ is shorthand for $\mathbf{if}\ b\ \mathbf{then}\ s\ \mathbf{else}\ \mathbf{skip}$, and $\mathbf{assert}\ \neg b$ is shorthand for $\mathbf{if}\ b\ \mathbf{then}\ \mathbf{abort}$. Then, the semantics of a **while** loop is:

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

The sequence is increasing and bounded, so the limit is defined.

Termination and preservation of weight. In our sub-distrubtion semantics, the probability of non-termination is $1 - \mathrm{wt}(\mu)$. A statement s is *lossless* iff for every sub-distribution μ, $\mathrm{wt}([\![s]\!]_\mu) = \mathrm{wt}(\mu)$. Programs that are not lossless strictly reduce the sub-distribution weight, and are called *lossy*. We consider two notions of termination for loops and we will show how to enforce them with a proof system (§ 6). A loop $\mathbf{while}\ b\ \mathbf{do}\ s$ is:

- *certainly (c.) terminating* if there exists N such that for every sub-distribution μ:

  $$\mathrm{wt}([\![\mathbf{while} b \mathbf{do} s]\!]_\mu) = \mathrm{wt}([\![(\mathbf{if} b \mathbf{then} s)^N; \mathbf{assert}\ \neg b]\!]_\mu).$$

  This is sufficient to ensure that the semantics of the loop coincides with the semantics of its N-th iterate.
- *almost surely (a.s.) terminating* if it is lossless.

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 does not entail losslessness.

4. Assertions

Now that we have seen the programs, we turn next to assertions on programs. The assertion language is the key to the expressivity of Ellora, allowing the user to state complex invariants involving probability and expectation. It also forms the foundation for the probability theory libraries we will see in the next section, which define various properties in terms of the assertion language.

For simplicity, we present a simplified, core version of the assertion language. Full Ellora provides a higher-order logic, with support for user-defined predicates and additional big operations. As outlined in Figure 3, the core logic consists of *S-assertions* and *P-assertions*, interpreted over states and probabilistic states respectively. We introduce the two classes of assertions and then follow with their semantics.

State layer. State assertions are formulas over *state expressions* $\tilde{e}$, and predicate over elements of State. They are built from program expressions e and two classes of variables only present in assertions: *logical variables* $\dot{y}$ which are bound by quantifiers and big operators, and *integral variables* $\hat{t}$ which are bound by integrals.

State expressions can be combined by *big operators* which have the form $\mathcal{O}_{\{\dot{y} \mid \phi\}} \tilde{e}$, where $\mathcal{O}$ is a "big" version of a commutative and associative binary operation with a neutral

$$
\begin{array}{lcl}
[\![\mathbf{skip}]\!]_\mu & = & \mu \\
[\![\mathbf{abort}]\!]_\mu & = & \mathbf{0} \\
[\![x := e]\!]_\mu & = & \mathsf{Mlet}\ m = \mu\ \mathsf{in}\ \mathsf{unit}\ m[x := [\![e]\!]_m] \\
[\![x \xleftarrow{\$} g]\!]_\mu & = & \mathsf{Mlet}\ m = \mu\ \mathsf{in}\ \mathsf{Mlet}\ v = [\![g]\!]_m\ \mathsf{in}\ \mathsf{unit}\ m[x := v] \\
[\![s_1; s_2]\!]_\mu & = & [\![s_2]\!]_{[\![s_1]\!]_\mu} \\
[\![\mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2]\!]_\mu & = & \mathsf{Mlet}\ m = \mu\ \mathsf{in}\ (\mathsf{if}\ [\![e]\!]_m\ \mathsf{then}\ [\![s_1]\!]_{(\mathsf{unit}\ m)}\ \mathsf{else}\ [\![s_2]\!]_{(\mathsf{unit}\ m)}) \\
[\![\mathbf{while}\ e\ \mathbf{do}\ s]\!]_\mu & = & [\![\mathbf{if}\ e\ \mathbf{then}\ s; \mathbf{while}\ e\ \mathbf{do}\ s]\!]_\mu
\end{array}
$$

Figure 2. Equational theory of programs

$$
\begin{array}{lll}
v ::= \dot{y} \mid \hat{t} & & \text{(Extended variables)} \\
\tilde{e} ::= e \mid v \mid \mathbf{1}_\phi \mid \tilde{e} + \tilde{e} \mid \tilde{e} \times \tilde{e} \mid \sum_{\{\dot{y}|\phi\}} \tilde{e} \mid \prod_{\{\dot{y}|\phi\}} \tilde{e} & & \text{(S-expr.)} \\
\phi ::= \tilde{e} \bowtie \tilde{e} \mid FO(\phi) & & \text{(S-assn.)} \\
p ::= \oint_\Gamma \tilde{e} \mid p + p \mid p \times p \mid \sum_{\{\dot{y}|\phi\}} p \mid \prod_{\{\dot{y}|\phi\}} p & & \text{(P-expr.)} \\
\eta ::= p \bowtie p \mid FO(\eta) \mid \eta \oplus \eta & & \text{(P-assn.)}
\end{array}
$$

Figure 3. Assertion syntax

element—think Σ for $+$, or Π for $\times$. The bound logical variable $\dot{y}$ represents the index, which ranges over the natural numbers satisfying the state assertion ϕ; we require this filter to be deterministic and true for at most finitely many indices. State expressions also contain characteristic functions $\mathbf{1}_\phi$ of state assertions ϕ, which take a state m and return 1 if ϕ holds on m and 0 otherwise.

State assertions ϕ are built from atomic state assertions using the usual connectives and quantifiers of first-order logic ($FO(\phi)$ in the syntax). Atomic state assertions are well-typed applications of predicates to state expressions; in the figure, we only consider binary predicates $\bowtie$, typically $<$ and $=$.

Probabilistic layer. Since our program state is represented by a sub-distribution $\mu \in \mathsf{pState}$, we need assertions that predicate over elements of pState. These are *probability assertions*, or *P-assertions*, and they form the second assertion layer. P-assertions express pre- and post-conditions in our program logic.

We begin by defining the probabilistic counterpart of state expressions, which we call *probability expressions* p. These are generalized polynomial expressions (built using constants, addition, multiplication, and their corresponding big operators) over *integral expressions* $\oint_\Gamma \tilde{e}$, where $\tilde{e}$ is a state expression, and Γ is list of pairs $(\hat{t}, g)$ binding integration variables to distribution expressions. Integral expressions calculate the expected value of $\tilde{e}$ on the state, where integration variables $\hat{t}$ are drawn from their corresponding distribution g. When Γ is empty, we will write the integral expression as simply $\oint \tilde{e}$.

$$
\begin{array}{lcl}
[\![\dot{y}]\!]^\rho_m & \triangleq & \rho(\dot{y}) \\
[\![\mathbf{1}_\phi]\!]^\rho_m & \triangleq & \mathbf{1}_{[\![\phi]\!]^\rho_m} \\
[\![\sum_{\{\dot{y}|\phi\}} \tilde{e}]\!]^\rho_m & \triangleq & \sum_{\{t | [\![\phi]\!]^\rho_m\}} [\![\tilde{e}]\!]^{\rho[\dot{y}:=t]}_m \\
[\![o(\tilde{e})]\!]^\rho_m & \triangleq & o([\![\tilde{e}]\!]^\rho_m) \\
\\
[\![\tilde{e}_1 \bowtie \tilde{e}_2]\!]^\rho_m & \triangleq & [\![\tilde{e}_1]\!]^\rho_m \bowtie [\![\tilde{e}_2]\!]^\rho_m \qquad \bowtie \in \{=, <\} \\
[\![FO(\phi)]\!]^\rho_m & \triangleq & FO([\![\phi]\!]^\rho_m) \\
\\
[\![\oint_\Gamma \tilde{e}]\!]^\rho_\mu & \triangleq & \sum_m \sum_{\overline{t_g}} [\![\tilde{e}]\!]^\rho_m \prod_{(-,g)\in\Gamma} [\![g]\!]^\rho_m(t_g)\ \mu(m) \\
[\![o(p)]\!]^\rho_\mu & \triangleq & o([\![p]\!]^\rho_\mu) \\
\\
[\![p_1 \bowtie p_2]\!]^\rho_\mu & \triangleq & [\![p_1]\!]^\rho_\mu \bowtie [\![p_2]\!]^\rho_\mu \qquad \bowtie \in \{=, <\} \\
[\![\eta_1 \oplus \eta_2]\!]^\rho_\mu & \triangleq & \exists \mu_1, \mu_2, \quad \mu = \mu_1 + \mu_2 \wedge [\![\eta_1]\!]^\rho_{\mu_1} \wedge [\![\eta_2]\!]^\rho_{\mu_2} \\
[\![FO(\eta)]\!]^\rho_\mu & \triangleq & FO([\![\eta]\!]^\rho_\mu)
\end{array}
$$

Figure 4. Semantics of assertions (excerpt)

Probabilistic assertions η are built from assertions on probabilistic expressions using the usual connectives and quantifiers, and a binary connective $\oplus$ called *split*. Informally, a probabilistic state μ satisfies the assertion $\eta_1 \oplus \eta_2$ if μ can be split as $\mu = \mu_1 + \mu_2$ such that μ_1 and μ_2 satisfy η_1 and η_2 respectively; we use this connective to model the split control flow for branching commands.

Semantics of assertions. The interpretation of logical variables is given by a *logical valuation* ρ mapping logical variables to values, while the interpretation of program variables depends on the assertion layer. S-expressions and S-assertions are interpreted in a state. Accordingly, their interpretations $[\![\tilde{e}]\!]^\rho_m$ and $[\![\phi]\!]^\rho_m$ are parametrized by a state m. S-expressions are interpreted as values, while S-assertions are interpreted as booleans. P-expressions and P-assertions are interpreted in a probabilistic state; their interpretations $[\![p]\!]^\rho_\mu$ and $[\![\eta]\!]^\rho_\mu$ are parametrized by a *probabilistic state* μ. P-expressions are interpreted as real numbers; P-assertions are interpreted as booleans. The validity of a P-assertion η in μ with logical variables ρ is denoted $\mu; \rho \models \eta \triangleq [\![\eta]\!]^\rho_\mu = \top$.

An excerpt of the semantics for assertions (along with state and probability expressions) is presented in Figure 4. We

highlight the atomic P-expressions, the *integral expressions* $\oint_\Gamma \tilde{e}$. When interpreted in a probabilistic state μ, the outer sum is a sum over all memories $m \in \mathsf{State}$, weighted by $\mu(m)$. Inside the outer sum, we create one integral variable t_g for each binding $(t, g) \in \Gamma$, and we compute a weighted sum over all t_g with t_g ranging over all values in the ground type of distribution g weighted by $g(t_g)$. We require all sums to be well-defined and finite.

Probability, expectation and necessity. The assertion logic is very powerful and is sufficient to define standard concepts from probability theory. Integral expressions can represent the probability of state formulas: the quantity $\oint_\Gamma \mathbf{1}_\phi$ interpreted in some probabilistic state μ is simply the probability that ϕ holds when for each $(t, g) \in \Gamma$, t is drawn from g, and when the program variables are drawn according to the distribution μ. Recall that we will use the notations

$$\Pr[\phi] \triangleq \oint \mathbf{1}_\phi \quad \text{and} \quad \mathbb{E}[\tilde{e}] \triangleq \oint \tilde{e}.$$

These are the usual definitions of probability and expectation, but for sub-distributions. For instance, $\Pr[\top]$ is not always interpreted as 1—in general, it is the total weight of the probabilistic state, which can range from 0 to 1. However, we continue to have $\Pr[\phi] + \Pr[\neg\phi] = \Pr[\top]$ for every state assertion ϕ in any probabilistic state.

Frequently, we want to assert that a sub-distribution is a proper distribution with weight 1: $\mathcal{L} \triangleq \Pr[\top] = 1$. We also often want to assert that a state formula ϕ holds on all possible terminating executions: $\Box\phi \triangleq \Pr[\neg\phi] = 0$. This $\Box$ operator is similar to a necessity operator in modal logic.

5. Libraries for probability theory

When proving properties of randomized algorithms, theorems from probability theory are an invaluable tool. Accordingly, a central part of Ellora is a collection of libraries defining high-level assertions for mathematical tools like expected value and independence of random variables, and supplying formalized versions of associated theorems. This toolbox allows complex reasoning in the assertion logic, especially useful for reasoning steps that aren't closely tied to the program code. Ellora builds on a wide variety of useful libraries, for instance for containers (e.g., sets and maps), base types (e.g., real numbers), derived types (e.g., bitstrings), and pure mathematics (e.g., basic real analysis and group theory). The library for probabilty theory is new, and we will focus on three useful tools. Simple versions of the definitions can be expressed in the core logic, while the most general definitions require the higher-order logic of full Ellora.

Distribution shapes. When analyzing a random draw from a distribution, a fundamental piece of information is the *shape* of the distribution—e.g., whether it is a coin flip distribution or a normal distribution—along with associated parameters like the probability of flipping heads, or the mean and variance. We can model this information with shape assertions: $x \sim g$ states that x is distributed according to the distribution expression g. For example, we can record that a variable has a Bernoulli (coin flip) distribution with bias w:

$$x \sim \mathbf{Bern}(w) \triangleq \Pr[x] = w \wedge \Pr[\neg x] = 1 - w,$$

or a geometric distribution with infinite support:

$$x \sim \mathbf{Geom}(w) \triangleq \forall \dot{y} \in \mathbb{N}.\ \Pr[x = \dot{y}] = (1 - w)^{\dot{y}} w.$$

Shape assertions also provide a convenient hook where we can access with facts about the distribution, like its range:

$$\tilde{e} \sim \mathbf{Unif}(0, 1) \Rightarrow \Box(0 \leq \tilde{e} \leq 1),$$

or its expectation and variance:

$$\tilde{e} \sim \mathbf{Geom}(c) \Rightarrow \mathbb{E}[\tilde{e}] = 1/c \wedge \mathbb{E}[\tilde{e} \cdot \tilde{e}] = (1 - c)/c^2.$$

Independence. As we saw in the analysis of hypercube routing, a powerful tool for analyzing randomized algorithms is *independence of random variables*. Informally, a collection of samples are independent if they are drawn from distinct sources of randomness. To reduce notation, we will write $\bar{x}$ for a list of $\{x_i\}$, and $\overline{x = y}$ to mean that the two lists are the same length, and $x_i = y_i$ for each index i. Then, we define:

$$\#\langle x_1, \ldots, x_n \rangle \triangleq \forall \overline{a \in \mathbb{Z}}.$$
$$(\Pr[\top])^{n-1} \Pr[\overline{x = a}] = \Pr[x_1 = a_1] \times \cdots \times \Pr[x_n = a_n].$$

This is the textbook definition of independence, plus a normalization term depending on $\Pr[\top]$ for sub-distributions.

We can also reason about independence between groups of random variables, even if the variables inside a group may not be independent:

$$\langle \overline{x} \rangle \ \# \ \langle \overline{y} \rangle \triangleq \forall \overline{a \in \mathbb{Z}}, \overline{b \in \mathbb{Z}}.$$
$$\Pr[\top] \Pr[\overline{x = a} \wedge \overline{y = b}] = \Pr[\overline{x = a}] \Pr[\overline{y = b}].$$

These definitions of independence extend naturally to non-integer variables.

This classical definition is quite low level, awkward for proving more general properties of independent random variables. For this reason, Ellora provides a second, more abstract definition of independence.[1] This definition relies on Ellora's axiomatization of discrete distributions, along with its theory for the monadic operations on distributions:

```
op dunit : 'a → 'a distr
op dbind : 'a distr → ('a → 'b distr) → 'b distr
```

We can define marginalization (projection) and product distributions in terms of these operations:

[1] Ellora provides a third definition of independence for lists of heterogenous random variables even though Ellora is not dependently typed; the encoding is slick but rather technical, so we omit the details here.

```
op dprj (d:'a distr) (f : 'a → 'b) : 'b distr =
  dbind d (dunit ∘ f)
op dprd (d1:'a distr) (d2:'b distr) : ('a*'b) distr =
  dbind d1 (fun a ⇒ dbind d2 (fun b ⇒ dunit (a, b))
```

allowing a simple definition of `eindep`: independence in terms of equality of distributions (denoted by ==).

```
op fpair (X:mem→'a) (Y:mem→'b) (m:mem) = (X m, Y m)
op eindep (d:mem distr) (X:mem → 'a) (Y:mem → 'b) =
  (dprj d (fpair X Y)) == dprd (dprj d X) (dprj d Y)
```

(For simplicity, we omit the scaling for sub-distributions.)

We prove equivalence between the abstract definition and the pointwise definition of independence above, and use the abstract definition to prove general facts about independence. For instance, we can permute and project independence:

$$\langle \overline{x} \rangle \# \langle \overline{y} \rangle \Rightarrow \langle \overline{y} \rangle \# \langle \overline{x} \rangle \quad \text{and} \quad \overline{x'} \subset \overline{x} \wedge \#\langle \overline{x} \rangle \Rightarrow \#\langle \overline{x'} \rangle;$$

independence is preserved by deterministic functions:

$$\#\langle \overline{x}, \overline{y} \rangle \Rightarrow \#\langle f(\overline{x}), g(\overline{y}) \rangle;$$

and expectation commutes with multiplication for independent variables:

$$\#\langle x, y \rangle \Rightarrow \Pr[\top] \cdot \mathbb{E}[x \cdot y] = \mathbb{E}[x] \cdot \mathbb{E}[y].$$

Anticipating the next section, our program logic features introduction rules for sampling commands to link independence to the concrete program. The first rule reflects the intuition behind independence—if variables $\overline{x}$ are independent, then a new sample is also independent of $\overline{x}$:

$$\textsc{Ind}\ \frac{x^* \notin \overline{x}}{\{\#\langle \overline{x} \rangle\}\ x^* \xleftarrow{\$} \mathcal{D}\ \{\#\langle \overline{x}, x^* \rangle\}}.$$

The second rule generalize to parameterized distribution:

$$\textsc{Ind-G}\ \frac{x^* \notin \overline{x}}{\{\#\langle \overline{x} \rangle \wedge \langle \overline{x} \rangle \# \langle \overline{x'} \rangle\}\ x^* \xleftarrow{\$} \mathcal{D}(\overline{x'})\ \{\#\langle \overline{x}, x^* \rangle\}}.$$

That is, if $\overline{x}$ are independent and independent of the parameters $\overline{x'}$, then a sample from $\mathcal{D}(\overline{x'})$ is independent from $\overline{x}$.

Concentration bounds. A particularly common tool in analyzing probabilistic algorithms is applying *concentration bounds*. Roughly, these theorems state that the sum of independent random samples should be close to the expectation. While some random samples may be larger than the mean, other random samples should be smaller than the mean. Thus, these errors should cancel out if we take many samples.

We have formalized the *multiplicative Chernoff bound*, a bound for sums of independent $0/1$ variables. In Ellora:

```
lemma Chernoff (d:mem distr) (Xi:int → mem → bool)
  (n:int) (μ δ:real) :
  let X = Σ [0, n) Xi in
  0 ≤ n ⇒ 1 < δ ⇒ Pr[⊤ | d] = 1
  ⇒ E[X | d] ≤ μ ⇒ indep [0, n) Xi d
  ⇒ Pr[X > (1 + δ)μ | d] ≤ (e^δ/(1 + δ)^(1+δ))^μ
```

The proof is rather involved, and relies on another common tool called *Markov's inequality*, which bounds the probabilitiy of a large deviation in terms of the expected value:

```
lemma Markov (d:mem distr) (X:mem → real) (a:real) :
  0 < a ⇒ □[0 ≤ X | d]
  ⇒ Pr[X ≥ a | d] ≤ E[X | d] / a
```

Ellora includes a complete formalization of both results.

6. Proof system

To connect the assertions to the program, Ellora includes a program logic. Judgments are of the form $\{\eta\}\ s\ \{\eta'\}$, where η and η' are P-assertions.

Definition 1. *A judgment* $\{\eta\}\ s\ \{\eta'\}$ *is* valid*, written* $\models \{\eta\}\ s\ \{\eta'\}$*, iff* $[\![s]\!]_\mu; \rho \models \eta'$ *for every probabilistic state* μ *and logical valuation* ρ *such that* $\mu; \rho \models \eta$*.*

6.1 Non-looping constructs

Figure 5 gathers the rules for non-looping constructs, we omit the standard structural rules. The rules for **skip**, assignments 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 rule for random assignment is a generalization of the usual rule for deterministic assignment, using a probabilistic substitution operator $\mathcal{P}$. Informally, $\mathcal{P}^g_x(\eta)$ replaces all occurrences of x in η with a new integration variable $\hat{t}$, and records that $\hat{t}$ should be drawn distributed according to g. Formally, $\mathcal{P}^g_x(\eta)$ is defined as a substitution on η. For integrals,

$$\mathcal{P}^g_x\left(\oint_{\Gamma'} \tilde{e}\right) = \oint_{(\hat{t},g)::\Gamma'[\hat{t}/x]} \tilde{e}[\hat{t}/x].$$

Note that we do not perform the substitution in the distribution g. Moreover, $\mathcal{P}$ acts as the identity on constants.

The rule for conditionals is unusual in that the post-condition must be of the form $\eta_1 \oplus \eta_2$; this reflects the semantics of a conditional statement, which first splits the initial probabilistic state depending on the guard, runs both branches, and recombines the results.

6.2 Loops

A well-known issue with probabilistic programs is that the standard Hoare rule for **while** loops is unsound. For instance, the judgment $\{\square\top\}$ **while** true **do skip** $\{\square\top\}$ is not valid, although $\{\square\top\}$ **skip** $\{\square\top\}$ is valid. In order to recover soundness, our rules for **while** loops (Figure 6) constrain the termination behavior of the loop and the assertions used for the loop invariant, using two kinds of side-conditions.

Probabilistic variants. When proving termination of deterministic programs, a typical tool is to find a *variant*—an expression in the language—that decreases by a fixed amount every iteration, with the loop exiting when the measure reaches 0. The situation is more complicated in the probabilistic case, since a loop with a probabilistic guard may

$$\frac{}{\{\eta\}\ \mathbf{skip}\ \{\eta\}} \qquad \frac{}{\{\eta[x := e]\}\ x := e\ \{\eta\}}$$

$$\frac{}{\{\eta\}\ \mathbf{abort}\ \{\Box\bot\}} \qquad \frac{}{\{\mathcal{P}^g_x(\eta)\}\ x \xleftarrow{\$} g\ \{\eta\}}$$

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

$$\frac{\{\eta_1\}\ s_1\ \{\eta_1'\} \qquad \{\eta_2\}\ s_2\ \{\eta_2'\}}{\{(\eta_1 \wedge \Box e) \oplus (\eta_2 \wedge \Box\neg e)\}\ \mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2\ \{\eta_1' \oplus \eta_2'\}}$$

Figure 5. Core rules: non-looping constructs

not have a measure that always decreases. Nonetheless, we can use two probabilistic versions of the variant:

1. *Deterministic variant.* The first kind of variant decreases deterministically each iteration, just like variants for deterministic program. This kind of variant proves certain termination of loops if the variant is initially bounded above, and forms part of rule [WHILE-C].

2. *Bounded variant.* The second kind of variant decreases probabilistically, but with probability at least $\epsilon > 0$. When the variant does not decrease, it can either stay constant or increases. We require that throughout all iterations, the variant must be bounded above by some fixed constant K. Both ϵ and K are fixed constants for the loop. This variant ensures a.s. termination of loops, and forms part of rule [WHILE-PVB].

These variants are reflected in the premises of our rules for loops (Figure 6). Ellora supports additional loop rules for programs that terminate almost surely but have unbounded variant, and programs that do not terminate with probability 1; we omit these rules here.

Closedness properties. Besides termination, we require the loop invariant to satisfy certain *closedness properties*. Closedness is a predicate on assertions useful for the soundness of the **while** rules. It guarantees that the invariant is preserved under the limit construction used to interpret **while** loops.

We consider here the case where the loop is lossless (the lossy case is in the supplemental material). If the **while** loop is lossless, its semantics is the limit of its non-truncated iterates:

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

Intuitively, this is because the assert statement for the nth truncated iterate filters out executions that have not terminated after n steps, but the weight of such executions tends to 0 as n grows since the **while** loop is lossless.

Now assume that $\models \{\eta\}\ \mathbf{if}\ b\ \mathbf{then}\ s\ \{\eta\}$. Then for every $n \in \mathbb{N}$, we also have $\models \{\eta\}\ (\mathbf{if}\ b\ \mathbf{then}\ s)^n\ \{\eta\}$. In order to conclude that $\models \{\eta\}\ \mathbf{while}\ b\ \mathbf{do}\ s\ \{\eta\}$, it is therefore sufficient to know that η is stable under limits. This precisely corresponds to the notion of *topological closedness*, which we require for our rules for almost-surely terminating loops ([WHILE-PVB] and [WHILE-PVU]).

Definition 2. *An assertion η is t-*closed *iff for every sequence $(\mu_n)_{n\in\mathbb{N}}$ of sub-distributions and logical valuation ρ s.t. $\lim_{n\to\infty} \mu_n = \mu$, if $\mu_n; \rho \models \eta$ for all $n \in \mathbb{N}$ then $\mu; \rho \models \eta$.*

We establish the closedness side-conditions via syntactic conditions, deferred to the supplemental material.

Proof rules. The rule [WHILE-C] (Figure 6) applies to certainly terminating loops, by requiring a state expression $\tilde{e}$ that certainly decreases at each iteration until it is 0, when the guard of the loop must be false. There is no requirement on the closedness of the loop invariant.

The rule [WHILE-PVB] (Figure 6) requires that the loop invariant is t-closed, and enforces a.s. termination of the loop. Informally, it assumes that the variant $\tilde{e}$ is (B)ounded and decreases with strictly positive probability; the amount of decrease ϵ is a strictly positive real constant.

6.3 Soundness

We can now prove soundness of our logic; details are in the supplemental material.

Theorem 1 (Soundness)**.** *Every judgment $\{\eta\}\ s\ \{\eta'\}$ provable using the rules of our logic is valid.*

Proof. By induction on the derivation of $\{\eta\}\ s\ \{\eta'\}$. □

7. Implementation and evaluation

We have built a prototype implementation of Ellora on top of WizWoz [name blinded], a tool-assisted framework for cryptographic proofs. WizWoz combines the benefits of proof assistants and program verifiers by letting users invoke external tools, like SMT-solvers, at any point during interactive, tactic-based proofs. Moreover, WizWoz provides a mature set of libraries (for sets, maps, lists, arrays, etc.), which we extended with new libraries for probability theory.

We verified all the programs presented in § 8 as well as some additional examples from the randomized algorithm literature (such as polynomial identity test, private running sums, properties about random walks, etc.) with our prototype. The verified proofs follow the corresponding proofs quite closely. The length of these paper proofs varies from few lines (like in the case of vertex-cover) to a page (like in the case of hypercube routing). We summarize in Table 1 the length of the verified program (LC) and the length of the formal proof (FPLC) for each example.

Ellora, WizWoz, and examples are available at [url blinded].

8. Examples

In this section, we will demonstrate Ellora on a selection of examples (more examples are available in the supplemental material). Together, they exhibit a wide variety of different proof techniques and reasoning principles, while demonstrating various uses of randomization in algorithmic design.

GENERIC RULE:

$$\textsc{While-X}\ \frac{\{\eta\}\ \mathbf{if}\ b\ \mathbf{then}\ s\ \{\eta\} \qquad \mathfrak{C}_{\mathbf{X}}}{\{\eta\}\ \mathbf{while}\ b\ \mathbf{do}\ s\ \{\eta \wedge \Box\neg b\}}$$

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

SIDE CONDITIONS:

$$\mathfrak{C}_{\mathbf{PVB}} \triangleq \{\mathcal{L} \wedge \Box(\tilde{e} = k \wedge 0 < k \leq K \wedge b)\}\ s\ \{\mathcal{L} \wedge \Box(0 \leq \tilde{e} \leq K) \wedge \Pr[\tilde{e} < k] \geq \epsilon\} \quad \tilde{e} : \mathbb{N}$$
$$\models \eta \Rightarrow \Box(0 \leq \tilde{e} \leq K \wedge \tilde{e} = 0 \Rightarrow \neg b)$$
$$\models \eta \ \text{tclosed}$$

Figure 6. Core rules: loops

Example	LC	FPLC
hypercube routing	100	691
coupon collector	27	270
vertex cover	30	70
pairwise-independent bits	30	240
private running sums	22	80
polynomial identity testing	22	45
random walk	16	50
dice sampling	10	55
matrix product testing	20	85

Table 1. Benchmarks

8.1 Randomization for approximation: vertex cover

We begin with a classical application of randomization: *approximation algorithms* for computationally hard problems. For problems that take long time to solve in the worst case, we can sometimes devise efficient algorithms that find a solution that is "nearly" as good as the true solution.

Our first example illustrates a famous approximation algorithm for the *vertex cover* problem. The input is a graph described by vertices V and edges E. The goal is to output a vertex cover: a subset $C \subseteq V$ such that each edge has at least one endpoint in C, and such that C is as small as possible.

It is known that this problem is NP-complete, but there is simple randomized algorithm that returns a vertex cover that is at on average at most twice the size of the optimal vertex cover. The algorithm proceeds by maintaining a current cover (initially empty) and considering each edge in order. If neither endpoint is in current cover, the algorithm adds one of the two endpoints uniformly at random. The Ellora program is:

```
proc VC (E : set<edge>) :
var   set<node> C = ∅;
for (e1,e2) in E do
  if (e1 ∉ C) ∧ (e2 ∉ C) then
    b ←$ {0,1};
    C ← (b ? e1 : e2) ∪ C;
  fi
end
```

Here, we represent edges as a finite set of pairs of nodes. We loop through the edges, adding one point of each uncovered edge to the cover `C` uniformly at random. The operator `b ? e1 : e2` returns `e1` if `b` is true, and `e2` if not.

To prove the approximation guarantee, we first assume that we have a set of nodes $\mathtt{C}^*$. We only assume that $\mathtt{C}^*$ is a valid vertex cover; i.e., each edge has at least one endpoint in $\mathtt{C}^*$. Then, we use the following loop invariant:

$$\mathbb{E}[\mathtt{size}(\mathtt{C} \setminus \mathtt{C}^*)] \leq \mathbb{E}[\mathtt{size}(\mathtt{C} \cap \mathtt{C}^*)]. \tag{1}$$

Given the loop invariant, we can prove the conclusion by letting $\mathtt{C}^*$ be the cover OPT of minimal size, and reasoning about intersections and differences of sets.

Clearly the invariant is initially true. To see why the invariant is preserved, let `e` be the current edge, with both endpoints out of `C`. Since $\mathtt{C}^*$ is a vertex cover, it has at least one endpoint of `e`. Since our algorithm includes an endpoint of `e` uniformly at random, the probability we choose a vertex not in $\mathtt{C}^*$ is at most $1/2$, so the expectation on the left in Equation (1) increases by at most $1/2$. If `e` is not covered in `C` but is covered by $\mathtt{C}^*$, there is at least a $1/2$ probability that we increase the intersection $\mathtt{C} \cap \mathtt{C}^*$, so the right side in eq. (1) increases by at least $1/2$. Thus, the invariant is preserved, and we can prove

$$\{\mathrm{isVC}(\mathtt{C}^*, \mathtt{E})\}\ \mathtt{VC(E)}\ \{\mathbb{E}[\mathtt{size}(\mathtt{C} \setminus \mathtt{C}^*)] \leq \mathbb{E}[\mathtt{size}(\mathtt{C} \cap \mathtt{C}^*)]\}$$

and by reasoning on intersection and difference of sets,

$$\{\mathrm{isVC}(\mathtt{C}^*, \mathtt{E})\}\ \mathtt{VC(E)}\ \{\mathbb{E}[\mathtt{size}(\mathtt{C})] \leq 2 \cdot \mathbb{E}[\mathtt{size}(\mathtt{C}^*)]\}.$$

8.2 Modeling infinite processes: the coupon collector

Our second example is the *coupon collector* process. The algorithm draws a uniformly random coupon on each day, terminating when it has drawn at least one of each kind of coupon. Our goal is to bound the average number of steps we need before terminating. In Ellora,

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

```
    ct  ← ct + 1;
    cur ←$ Unif[N];
  end
  time[p] ← ct;
  cp[cur] ← 1;
  X ← X + time[p];
end
```

The code uses the array `cp` to keep track of the coupons seen so far. 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 `time[p]`, and the total number of steps in `X`.

The code involves two nested loops, and so we have two loop invariants. Handling the inner while loop has a probabilistic guard: every iteration, there is a finite probability that the loop terminations (i.e., if we draw a new coupon), but there is no finite bound on the number of iterations we need to run. We will show that we can apply rule [WHILE-PVB].

For the termination analysis, we use an 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 $\rho(\mathtt{p})$ of seeing a new coupon and decreasing the variant. Furthermore, the variant is bounded by 1, and the loop exits when the invariant reaches 0. So, the termination precondition of [WHILE-PVB] holds.

For the inner loop invariant η_{in}, we use the formula:

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

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 $\mathtt{ct} \geq c$, and the probability of stopping at c iterations is $(1-\rho(\mathtt{p}))^c\rho(\mathtt{p})$—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 $k = \mathtt{ct}$. If the loop terminated, it terminated before the current iteration: $\mathtt{ct} < k$. Furthermore, the probability of continuing at iteration k is $(1-\rho(\mathtt{p}))^k$.

At the end of the loop we have $\Box\mathtt{cp[cur]} = 0$. So, by the first conjunct and some manipulations,

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

holds when the inner loop exits, precisely describing the distribution of iterations `ct` as $\mathbf{Geom}(\rho(\mathtt{p}))$ 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 rule [WHILE-C]. For the loop invariant, we take:

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

The first conjunction states that the previous waiting times follow a geometric distribution with parameter $\rho(i)$; this assertion is verified by the previous reasoning on the inner loop. The second assertion asserts that we are keeping track of the total waiting time so far. The final two assertions state that there are at most $\mathtt{p}-1$ flags set in `cp`. Thus,

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

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:

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

8.3 Limited randomness: pairwise-independent bits

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.

If we have a collection of random variables X_i, we can express pairwise independence with the following assertion:

$$\forall i, j \in \mathbb{N}.\ i \neq j \Rightarrow X_i \,\#\, X_j.$$

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 magnify a small number of mutually independent bits into more pairwise-independent bits. Here is one procedure:

```
proc pwInd (N : int) :
var bool X[2^N], B[N];
for i = 1 to N do:
  B[i] ←$ Ber(1/2);
end
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] fi
  end
end
```

Above, $\oplus$ is the boolean XOR operation; `bits(j)` returns the set of bit positions that are set in the binary expansion of `j`.

Guaranteeing pairwise-independence requires delicate reasoning about the independence of random variables. Roughly,

we rely on a key fact about independence (which we fully verify): for a uniformly distributed random boolean random variable Y, and a random variable Z (of any type),

$$Y \mathbin{\#} Z \Rightarrow Y \oplus f(Z) \mathbin{\#} g(Z) \tag{2}$$

for any two boolean functions f, g. Then, note that

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

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 do not agree. 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, $\mathtt{X}[i] \mathbin{\#} \mathtt{X}[k]$ follows from eq. (2) taking Z to be the distribution on tuples $\langle \mathtt{B}[1], \ldots, \mathtt{B}[\mathtt{N}] \rangle$ excluding $\mathtt{B}[r]$. This verifies the claimed specification:

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

9. Related work

There is a long tradition of research in the formal verification of probabilistic programs. Early works by Kozen [17], Ramshaw [22], Reif [24], Sharir et al. [27] laid the groundwork for verification of randomized algorithms.

Program logics for probabilistic programs. In a series of works initiated by Morgan et al. [21] and described in their textbook [18], McIver, Morgan, and their collaborators develop deductive verification methods for programs written in pGCL, an imperative language with probabilistic choice and (demonic) non-determinism; there are many works building on top of this language [10, 11, 14, 16]. In pGCL, the semantics of programs is based on weakest preconditions, and assertions are interpreted as positive-real-valued functions over states. We use Hoare-style rules, but more importantly we use two different kinds of assertions interpreted as boolean-valued functions on states and probabilistic states respectively. Moreover, our language supports general distribution expressions while pGCL is limited to distributions built using probabilistic choice. The two-layer assertion language and the primitive support for distribution expressions are key for supporting expressive and concise assertions.

den Hartog [8] uses a logic similar pGCL (but in Hoare-style). Additionally, his **while** rule is based on a semantic closure condition. The use of semantic conditions is undesirable for verification, because it requires reasoning about the semantics of programs. Moreover, neither pGCL nor the logic by den Hartog [8] support big operations, another key ingredient to have expressive and concise assertions.

More recent works develop program logics for restricted settings. Chadha et al. [3] give a decidable Hoare logic similar to ours for a probabilistic language without **while** loops; decidability imposes strong restrictions on program values. Rand and Zdancewic [23] formalize a Hoare logic for probabilistic programs; their setting is more restrictive than ours, both for the assertion language and for the class of programs considered. In particular, they impose strong restrictions on **while** loops.

Formalizations of probability theory. Formalizations of measure and integration theory in general purpose interactive theorem provers were investigated in several works [6, 12, 13, 19, 25]. Audebaud and Paulin-Mohring [1] propose an axiomatic approach for discrete distributions, and use it for reasoning about functional probabilistic programs. These formalizations have been used to verify several case studies, but they all rely on general-purpose theorem provers. We focus on randomized algorithms, aiming for more concise and lightweight verification similar to paper proofs. None of these works formalize concentration bounds, but Avigad et al. [2] recently completed a proof of the Central Limit theorem, which is the principle underlying concentration bounds.

Other approaches. There have been many other significant works to verify probabilistic program using different formal approaches. Techniques range from model checking (see e.g., Katoen [15]) to abstract interpretation (see e.g., Cousot and Monerau [7], Monniaux [20]), etc. An elegant method based on martingales is used by Chakarov and Sankaranarayanan [4, 5] for inferring expectation invariants and other properties. Using their method, they 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 [9] for analyzing probablistic termination. Sampson et al. [26] use a mix of static and dynamic analysis to check probabilistic programs from the approximate computing literature.

10. Conclusion and perspective

We have developed and implemented a general verification platform for randomized programs, and shown the feasibility of proving examples with complex proofs. We believe our system is already powerful enough to contemplate formalization in many areas of theoretical computer science. Prime targets include accuracy and differential privacy of algorithms, lower bounds, and distributed algorithms.

We also view our work as an important step towards building a foundational system for reasoning about probabilistic programs based on a clear separation between the underlying proof assistant and the code for program logic. We envision an extensible system where the trusted computing base consists exclusively of a simple and readable checker for a lightweight higher-order logic, with rules for program logics built on top.

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

[4] A. Chakarov and S. Sankaranarayanan. Probabilistic program analysis with martingales. In N. Sharygina and H. Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013*, volume 8044 of *Lecture Notes in Computer Science*, pages 511–526. Springer, 2013.

[5] 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.

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

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

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

[15] J. Katoen. Perspectives in probabilistic verification. In *2nd IEEE/IFIP International Symposium on Theoretical Aspects of Software Engineering, TASE 2008*, pages 3–10. IEEE Computer Society, 2008.

[16] J. Katoen, A. McIver, L. Meinicke, and C. C. Morgan. Linear-invariant generation for probabilistic programs. In R. Cousot and M. Martel, editors, *Static Analysis - 17th International Symposium, SAS 2010*, volume 6337 of *Lecture Notes in Computer Science*, pages 390–406. Springer, 2010.

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

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

[19] 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.

[20] 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.

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

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

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

[24] J. H. Reif. Logics for probabilistic programming (extended abstract). In *12th ACM Symposium on Theory of Computing, STOC 1980*, pages 8–13. ACM, 1980.

[25] 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, TPHOLs 2004*, volume 3223 of *Lecture Notes in Computer Science*, pages 271–286. Springer, 2004.

[26] 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.

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

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

[29] 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.