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 verification: even intuitive properties of simple programs can have elaborate proofs, requiring intricate termination analyses, sophisticated tools from probability theory, or complex probabilistic invariants. We present a deductive verification platform for reasoning about general properties of probabilistic programs, including high-probability bounds on error and expected running time, in several steps.

First, we define an expressive assertion language which naturally captures notions such as probability, expectation, independence, and distribution laws. Second, we describe a program logic with a rich set of rules for reasoning about loops with different termination behaviors, and prove its soundness. Third, we propose new mathematical libraries for probability theory, which allow for a tight integration between the libraries and the program logic.

We realize our system in a prototype combining interactive verification (in the style of proof assistants) and automatic verification (in the style of program verifiers), and we demonstrate its utility by building machine-checked proofs for a broad collection of examples from the algorithms literature. The examples demonstrate different uses of randomization, diverse properties, and significantly different proof styles. Altogether, our work demonstrates that deductive verification of general randomized algorithms is practical, today.

1. Introduction

Randomized algorithms are one of the richest areas in algorithms research. Blessed with the power to draw random samples during computation, randomized algorithms are able to fulfill a world of guarantees that deterministic algorithms simply cannot achieve; examples include differential privacy and security properties like indistinguishability. Similarly, there are many problems where simple randomized algorithms achieve efficiency unmatched by known deterministic algorithms. Beyond computer science, randomized computations are an effective tool for modeling uncertainty in real-world settings, like distributed systems, biological processes, *etc*.

Verification of randomized algorithms is a vibrant field of research, with concentrated activity in model checking, program analysis, and interactive proofs. Research in model checking has arguably achieved the broadest impact, notably through tools and case studies—for instance, the website[1] of the probabilistic model checker PRISM lists about two dozen tools. In contrast, interactive proofs of randomized algorithms has received only scattered attention, resulting in a paucity of tools and verified examples.

Accordingly, many randomized algorithms simply cannot be formally verified today. On the one hand, model checking and program analysis tools favor automation at the expense of expressiveness, limiting the scope of verifiable properties, and in some cases only verifying specific instances of algorithms rather than the general algorithms that appear in the literature. On the other hand, prominent tools for interactive verification of probabilistic programs like EasyCrypt [4, 6] are sufficiently expressive but target *relational* (more precisely 2-safety) properties, offering poor support for more traditional properties.

The central challenges. Based on an examination of randomized algorithms from standard textbooks and papers in both programming languages and algorithms research, we have identified several main obstacles to verification.

1. Complex computations. Many probabilistic programs perform complex computations over mathematical objects (from discrete mathematics, algebra, geometry, *etc*). For instance, programs for sampling Gaussian distributions over integer lattices [16] rely on the Gram-Schmidt decomposition to compute an orthonormal basis on the Euclidean space $\mathbb{R}^n$; recent breakthroughs for computing discrete logarithms over fields of small characteristic [24] use the index calculus method; *etc*.

2. Rich properties. Randomized algorithms are designed for a wide range of purposes, and satisfy an extensive variety of properties. The most traditional properties are (i) *correctness*: the algorithm should compute the right answer; (ii) *precision*: the algorithm should have low probability of failure; and (iii) *computational efficiency*, expressed probabilistically or in terms of expected value. However, there is also mountain of domain-specific properties. For instance, proofs of cryptographic algorithms often establish that a "bad" event has low probability in a probabilistic experiment, or that the secret key is probabilistically independent from the view of the adversary.

In order to prove the target property, proofs of randomized algorithms also rely on intermediate properties to describe program invariants. Proofs may use higher-level properties from probability theory like independence of random variables and assertions about the distribution law of certain samples, or they may use non-probabilistic assertions to track intermediate computations.

3. The need for mathematical tools. Proofs of randomized algorithms commonly invoke advanced tools from probability theory. A typical example is a *concentration bound*, which provides an upper bound on the deviation of a sum of independent random variables from its expectation; accordingly, applying this kind of bound requires first stating and proving independence between program expressions, and getting a handle on the expected value of the sum. Proofs may also utilize domain-specific mathematical theorems from fields like algebra or analysis.

Contributions. We present an interactive verification platform for randomized algorithms, demonstrating its use by proving correctness or efficiency for a representative set of examples from first principles. In more detail, we make the following contributions.

1. A sound and usable program logic. We formalize an expressive program logic for pWhile, a core imperative language with probabilistic sampling, and prove its soundness. The language we choose is both general, in the sense that it does not impose any restriction on the way the control-flow of programs can depend on probabilistic values, and extensible, in the sense that the set of expressions can be tailored to examples.

We also base our development on a very rich assertion language, which can concisely encode useful notions and facts from probability theory. In particular, we use assertions for modeling properties of standard distributions, elementary properties of probability and expectation, probabilistic independence, and more. Notably, we make crucial use of *big operators*, which are critical for handling the complex invariants common in randomized algorithms.

Our proof system includes a rich set of rules, especially for handling **while** loops. Informally, each rule is an instance of the usual rule for loops, augmented with a pair of additional premises; typically, a first premise constrains the termination behavior of the

[1] See http://www.prismmodelchecker.org/.

loop, while the second premise constrains the assertion used as loop invariant. The two premises are closely linked, i.e., the restriction on loop invariants depends on the termination behavior of the loop. Our rules consider *certain* termination, *almost-sure* termination, and arbitrary termination. The rules are not mutually exclusive—for instance, the rule for arbitrary termination is convenient in the common case where we want to upper bound the probability of a bad event, regardless of the actual termination behavior of the loop.

2. A functional implementation. We have built a prototype, called Ellora, on top of the EasyCrypt proof assistant. Starting from EasyCrypt is beneficial for several reasons. First, EasyCrypt was designed specifically for verifying probabilistic programs, and features facilities to easily write probabilistic programs, reason about memories, apply Hoare rules, etc. Second, EasyCrypt offers an expressive higher-order logic for defining mathematical notions, and a mature set of libraries (for reals, integers, arrays, lists, *etc*). Third, EasyCrypt 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.

Conversely, Ellora includes two elements which could grow EasyCrypt. First, Ellora includes several new mathematical libraries, covering big operators, infinite series, and discrete probability, for instance. For probability theory in particular, our library is both more foundational and richer than the EasyCrypt library, which is axiomatic. Second, Ellora treats program state as first-class. As a consequence, assertions in Ellora are foundational and easily extensible—in EasyCrypt, definitions or constructs involving program state must be handled in the trusted computing base. Taken together, these elements have the potential to make EasyCrypt foundational, as we discuss in § 7.

3. Formalized verification of example algorithms. We formalize a representative set of examples, including approximation algorithms, property testing algorithms, and other examples from the cryptography and privacy literature. Taken together, our examples demonstrate that machine-checked proofs of randomized algorithms are now within reach.

Structure of the paper. To warm up, we begin by outlining the proof of the *coupon collector process*, demonstrating the chief difficulties in verifying randomized algorithms, and demonstrating our program logic in action (§ 2). Next, we describe the syntax and semantics of programs and assertions (§ 3 and 4) in our system. We follow with the core rules of the program logic (§ 5) and derived assertions and rules for reasoning about the key notions of independence and probability laws (§ 6). We conclude by detailing our implementation (§ 7) and several case studies (§ 8).

2. A motivating example

Suppose there are N types of coupons, and every day we receive a uniformly random coupon. On average, how many days will it take to collect at least one of each kind of coupon?

This process is known as the *coupon collector process*, a standard example of a randomized algorithm and a useful tool to model phenomena in domains from population genetics to cache scheduling. Verifying this example requires (i) performing precise reasoning in a general probabilistic language; (ii) expressing and reasoning about complex program invariants; and (iii) applying tools from probability theory.

The proof, on paper. We view the process as a sequence of *phases*, with each phase ending when a new coupon is collected. Abstracting a bit more, we can model each coupon draw as a coin flip. In the first phase, we receive a new coupon immediately. In the second phase, we repeatedly flip a biased coin that is heads with probability $(N-1)/N$ until we see a heads. In phase i, we flip a coin biased with probability $(N-i+1)/N$ until we see a heads, and so on. Our goal is to bound the average, or *expected*, number of coin flips before we stop.

On paper, we can analyze this process by describing the distribution of the waiting time for each phase. If we flip a coin with bias p until we see the first heads, the number of flips follows a distribution known as the *geometric distribution* with parameter p. By applying a probability theory fact about the geometric distribution, the expected waiting time is $1/p$ flips. Finally, we execute for N phases until we have collected all the coupons. By linearity of expectation, the average total time is the sum of the average times for each phase.

Coding the coupon collector. We now sketch the verification of this example. For now, we simplify aspects of the verification; we will fill in the details later in § 8. The first step is to express the waiting time of the process; we use the following program:

```
var int cp[N], t[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
  t[p] ← ct;
  cp[cur] ← 1;
  X ← X + t[p];
end
```

We remember which coupons we have seen so far in the `cp` array, and the time we waited for each of the past coupons in the `t` array; both arrays are initialized with all zeros. The outer **for** loop iterates over the number of coupons we have seen so far, where each iteration corresponds to one phase.

In each iteration, we first sample the current coupon from the uniform distribution on $\{1, \ldots, \mathtt{N}\}$, denoted by `Unif(N)`. Then, while the current coupon isn't new, we keep sampling coupons in the inner **while** loop while incrementing a counter `ct` to record the number of iterations. Once we see a new coupon, we exit the **while** loop, record the new coupon, and record how long we waited. Then, we move on to the next coupon.

Reasoning about loops and termination: the inner loop. We first focus on the inner **while** loop, which is the most difficult to verify. We will show that the waiting time stored in `t[p]` is distributed according to a geometric distribution with the parameter depending on the phase, i.e.,

$$\mathtt{t[p]} \sim \mathtt{Geom((N - p + 1)/N)}.$$

We begin the analysis with a termination analysis. Note that there is no constant bound on the number of iterations performed by the inner loop; for any natural number k, there is some small, but non-zero probability that we will draw k old coupons without seeing a new one. Nevertheless, there is some fixed, non-zero probability of exiting the loop at every iteration—here, every iteration we exit with probability $(\mathtt{N}-\mathtt{p}+1)/\mathtt{N}$. This fact is enough to prove that the loop terminates with probability 1—*almost-sure termination*.

Our verification uses a rule for reasoning about probabilistic **while** commands that terminate almost-surely. Roughly, we track the probability that the counter `ct` is equal to $1, 2, \ldots$ as we proceed through the loop, by computing the probability that we exit (i.e., we draw a new coupon) and the probability that we continue (i.e., we draw an old coupon). The key portion of the (simplified) loop invariant is:

$$\forall i \in \mathbb{N}.\ \Pr[\mathtt{ct} = i \wedge \mathtt{cp[cur]} = 0] = \left(\frac{\mathtt{p}-1}{\mathtt{N}}\right)^i \left(\frac{\mathtt{N}-\mathtt{p}+1}{\mathtt{N}}\right).$$

The formula $\mathtt{ct} = i \wedge \mathtt{cp[cur]}$ asserts that the guard is false and the counter is i—i.e., that we exit the loop on the ith iteration.

At the end of the loop, we know that $\Pr[\mathtt{cp[cur]} = 0]$ is equal to the probability of termination (exactly 1), so we can conclude

$$\forall i \in \mathbb{N}.\ \Pr[\mathtt{ct} = i] = \left(\frac{\mathtt{p}-1}{\mathtt{N}}\right)^i \left(\frac{\mathtt{N}-\mathtt{p}+1}{\mathtt{N}}\right).$$

With this assertion, we have completely characterized the distribution of the waiting time ct, which we store into t[p]. In our assertion logic, we can now wrap this fact into a more concise form by using the definition of geometric distribution:

$$\mathtt{t[p]} \sim \mathtt{Geom((N - p + 1)/N)}.$$

Reasoning about complex invariants: the outer loop. Now, we turn to the outer **for** loop. Here, since we know there are exactly N iterations, the loop and termination reasoning is more straightforward. The relevant part of the outer loop invariant is:

$$\forall i \in [\mathtt{p}-1].\ \mathtt{t}[i] \sim \mathtt{Geom}\left(\frac{\mathtt{N}-i+1}{\mathtt{N}}\right) \wedge \Box\Big(\mathtt{X} = \textstyle\sum_{j\in[\mathtt{p}-1]} \mathtt{t}[j]\Big),$$

where we use notation $[n] \triangleq \{1, \ldots, n\}$ in the sum index.

The first conjunct says that each waiting time $\mathtt{t}[i]$ is distributed as a geometric distribution with the given parameter, while the second conjunct states that the accumulator X is a sum of the waiting times; $\Box\phi$ means ϕ is true on all possible program executions. This sum is actually an example of a *big operator*, a concise and flexible way to reason about a group of expressions. These operators are useful for analyzing randomized algorithms, when the number of expressions we want to manipulate may be large, or not fixed statically.

Applying theorems: completing the reasoning. Now, we wish to bound the expected total waiting time X. This reasoning involves a few facts from probability theory. Our assertion language, with the help of some defined assertions, is rich enough to concisely state useful axioms from probability theory.

First, we apply a fact about linearity of expectations to deduce

$$\mathbb{E}[\mathtt{X}] = \mathbb{E}\left[\textstyle\sum_{i\in[\mathtt{N}]} \mathtt{t}[i]\right] = \textstyle\sum_{i\in[\mathtt{N}]} \mathbb{E}[\mathtt{t}[i]].$$

Then, we use a fact about the expected value of the geometric distribution:

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

thus concluding the proof.

3. Programs

3.1 Syntax

We base our theoretical development on pWhile, a core probabilistic imperative language that can conveniently capture a wide range of randomized algorithms. This language extends the While language with two constructs: $\mathbf{abort}$, 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:

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

where x, e, and g range over the sets $\mathcal{X}$ of variables, $\mathcal{E}$ of expressions and $\mathcal{D}$ of distribution expressions respectively. The set $\mathcal{E}$ is defined inductively from the set $\mathcal{X}$ and a set $\mathcal{F}$ of function symbols, while the set $\mathcal{D}$ 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; $\mathbf{Bern}(e)$, the Bernoulli distribution with parameter e, is a valid distribution expression.

We assume that expressions, distribution expressions, and statements are typed. Program variables and (standard) expressions are assigned *ground types*: either a primitive type (booleans, integers, *etc.*), or a type of the form $C\ \overline{T}$ where $\overline{T}$ is a list of ground types and C is a type constructor (like products, lists, or arrays, and other container types). In addition, we consider *distribution types* of the form $\mathbb{D}\ T$ where T is a ground type, and function types, of the form $T_1 \times \ldots \times T_n \to T$ where the T_is are ground types or distribution types. Base types are used for (standard) expressions, whereas distribution types are used for distribution expressions. As usual, each function symbol is given a function type, and we use a simple type system to ensure that functions are applied to arguments of the correct type.

The type system extends to statements in the expected way. For instance, an assignment $x := e$ is well-typed if x and e have the same type T, whereas a probabilistic assignment $x \xleftarrow{\$} g$ is well-typed if x has type T and g has type $\mathbb{D}\ T$.

3.2 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 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.

Next, we define the set State of states as well-typed finite maps from variables to values, where the set $\mathcal{V}$ of values is defined as $\bigcup_T [\![T]\!]$, with T ranging over all discrete types. By construction, the set State is countable. Finally, we define the set pState of probabilistic states simply as $\mathbf{Distr}$ State. Note that 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 statements is defined in two steps. We first define the semantics $[\![e]\!]_m$ of an expression $e \in \mathcal{E}$ and $[\![g]\!]_m$ of a distribution expression $g \in \mathcal{D}$. The semantics is parametrized by a state m, and is defined in the usual way; in particular, $[\![e]\!]_m \in [\![T]\!]$ when e has type T, and $[\![g]\!]_m \in \mathbf{Distr}\ [\![T]\!]$ when g has type $\mathbb{D}\ T$. For simplicity, we require all distribution expressions to be interpreted as proper distributions.

Now, we can define the semantics $[\![s]\!]_\mu$ of a statement. The semantics is parametrized by a probabilistic state μ, and is defined by the equations of Figure 1. The semantics of **while** loops can be defined as the least fixed point of a continuous and monotonic operator over pState. However, it is more convenient to make the construction explicit [3, 17]. 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 $\mathbf{if}\ b\ \mathbf{then}\ s\ \mathbf{else\ skip}$, and $\mathbf{assert}\ \neg b$ is shorthand for $\mathbf{if}\ b\ \mathbf{then\ abort}$. Then, the semantics of a **while** loop is the

limit of its truncated iterates:

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

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

In constrast, the limit of (non-truncated) iterates does not always exists. However, the limit of iterates exists and coincides with the limit of truncated iterates when the loop satisfies a *losslessness* property—we now turn to this property, which is related to termination behavior of the loop.

3.3 Termination and preservation of weight

For deterministic programs, program termination is binary: either a program terminates on its input, or it loops forever. When we introduce randomness into the program, the termination behavior may depend on random sampling, introducing new kinds of (non-)termination. 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 are now ready to introduce two notions of termination for loops. While the definitions are semantic in nature, we will later see how to enforce termination with a proof system (§ 5). We say that 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 more probabilistic in nature: the program always terminates, 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.

For a few examples of the different termination behaviors, **for** loops are certainly terminating; the bounded one-dimensional random walk is almost surely terminating, but it is not certainly terminating; random walks in dimension 3 and higher are lossy [30].

4. Assertions

Our assertion language, outlined in Figure 2, contains *S-assertions* and *P-assertions*, which are interpreted over states and probabilistic states respectively. We introduce the two classes of assertions and then follow with their semantics.

4.1 State layer

The first layer of assertions contains *state assertions*, or *S-assertions*, which predicate over elements of State.

State assertions are formulae over *state expressions* $\tilde{e}$. These expressions are built from program expressions e and two new classes of variables only present in assertions: *logical variables* $\dot{y}$ are bound by quantifiers and big operators, while *integral variables* $\hat{t}$ are bound by integrals.

State expressions can also be combined in two new ways. A *big operator* has the form $\mathcal{O}_{\{\dot{y}|\phi\}}\tilde{e}$, where $\mathcal{O}$ is a "big" version of a commutative and associative binary operation with a neutral 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).

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

Figure 2. Assertion syntax

State assertions ϕ are built from atomic state assertions using the usual connectives and quantifiers of first-order logic (denoted by $FO(\phi)$ 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 $=$.

4.2 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 frequently write the integral expression as $\oint\tilde{e}$.

Probabilistic assertions η are built from atomic probabilistic 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.

4.3 Semantics of assertions

Now, we turn to the semantics of expressions and assertions. The interpretation of logical variables is given by a logical valuation ρ that maps logical variables to values, while the interpretation of program variables depends on the assertion layer.

In the first 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.

In the probabilistic layer, 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. We will use notation when η is valid in μ with logical variables ρ:

$$\mu;\rho \models \eta \triangleq [\![\eta]\!]^\rho_\mu = \top.$$

An excerpt of the semantics for assertions (along with state and probability expressions) is presented in Figure 3. We highlight

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

Figure 1. Equational theory of programs

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

Figure 3. Semantics of assertions (excerpt)

the atomic P-expressions, the integral expressions with the form $\oint_\Gamma \tilde{e}$. When interpreted in a probabilistic state μ, the outer sum is a weighted sum over all memories $m \in \mathsf{State}$, weighted by $\mu(m)$. Inside the outer sum, we create one summation 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, and the weight being $g(t_g)$.

In general, there are distributions and expressions where the integral semantics is either infinite, or not even defined. So, we require all sums to be well-defined and finite for an assertion involving integral expressions to hold.

4.4 Probability, expectation and necessity

With our assertion logic, we can define assertions to make certain properties easier to express. We will discuss defined assertions further in § 6, but for now we introduce some simple abbreviations that will be used throughout the paper.

Integral expressions can represent the probability of state formulae: 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. For readability, we will use the notation:

$$\Pr[\phi] \triangleq \oint \mathbf{1}_\phi.$$

This expression is analogous to the usual definition of probability, 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 claim that a sub-distribution is a proper distribution with weight 1, which is captured by the following probabilistic assertion:

$$\mathcal{L} \triangleq \Pr[\top] = 1.$$

We also often want to state that a state formula ϕ holds on all possible terminating executions. This can be written

$$\Box\phi \triangleq \Pr[\neg\phi] = 0.$$

This $\Box$ modality is similar to a necessity operator in modal logic.

5. Proof system

We are now well-prepared for the core of our system: the program logic. We first introduce judgments and structural rules. Then, we present the main rules for program constructs, including a rich set of rules for loops. Finally, we discuss on useful derived rules.

5.1 Judgments and structural rules

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* $\mu; \rho \models \eta$*.*

Validity of judgments is preserved under standard structural rules, like the rule of consequence:

$$\textsc{Cons}\ \frac{\eta_0 \Rightarrow \eta_1 \qquad \{\eta_1\}\ s\ \{\eta_2\} \qquad \eta_2 \Rightarrow \eta_3}{\{\eta_0\}\ s\ \{\eta_3\}}$$

Besides serving its usual role, the rule of consequence serves as the interface between the program logic and theorems from mathematical and probability theory. For instance, we use this rule to apply concentration bounds and other mathematical theorems.

5.2 Non-looping constructs

Figure 4 gathers the rules for non-looping constructs. The rules for **skip**, assignments and sequences are all straightforward. The rule for **abort** requires $\Box\bot$ to hold after execution; this assertion uniquely characterizes the null sub-distribution.[2]

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 according to the distribution g. More formally, $\mathcal{P}^g_x(\eta)$ is defined as a substitution on η; the case for integrals is

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

[2] $\Box\bot$ should not be confused with the probabilistic assertion $\bot$, which is not satisfied by *any* sub-distribution.

$$\textsc{Skip}\ \frac{}{\{\eta\}\ \mathbf{skip}\ \{\eta\}}$$

$$\textsc{Abort}\ \frac{}{\{\eta\}\ \mathbf{abort}\ \{\Box\bot\}}$$

$$\textsc{Assgn}\ \frac{}{\{\eta[x := e]\}\ x := e\ \{\eta\}}$$

$$\textsc{Sample}\ \frac{}{\{\mathcal{P}_x^g(\eta)\}\ x \xleftarrow{\$} g\ \{\eta\}}$$

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

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

Figure 4. Core rules: non-looping constructs

Note that we do not perform the substitution in the new 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 $\eta_1 \oplus \eta_2$. Intuitively, the split post-condition (and pre-condition) 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.

5.3 Loops

A well-known issue with probabilistic programs is that the standard Hoare rule for **while** loops is unsound. For instance, the judgment $\{\top\}\ \mathbf{while}\ \text{true}\ \mathbf{do}\ \mathbf{skip}\ \{\top\}$ is not valid, although $\top$ is a valid invariant for $\mathbf{skip}$. In order to recover soundness, our rules for **while** loops (Figure 5) 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 not have a measure that always decreases. Nonetheless, we can use three probabilistic versions of the variant:

1. *Deterministic variant.* The first kind of variant decreases deterministically each iteration, just like variants for determinstic 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 kind of variant ensures a.s. termination of loops, and forms part of rule [WHILE-PVB].

3. *Unbounded variant.* The final kind of variant can be unbounded, but the probability of strictly decreasing must be at least the probability of staying constant or increasing. Furthermore, the variant can increase by at most 1 every iteration. This kind of variant is also sufficient for proving a.s. termination, but the argument is more subtle than in the other cases. This variant forms part of rule [WHILE-PVU].

Each variant is formalized with a logical assertion ensuring that (a) some measure decreases (probabilistically) at every iteration, and (b) the program exits the loop when the measure reaches 0.

Closedness properties. Besides termination, we require the loop invariant to satisfy certain *closedness properties*. Closedness is a predicate on assertions, and guarantees that the invariant is preserved under the limit construction used to interpret **while** loops. We give a brief intuition for these properties, and why they are sufficient for the soundness of the **while** rules.

We consider the two cases: either the loop is lossless, or it is lossy. If the **while** loop is lossless, its semantics can be given as 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 truncated iterate n 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 $\mu; \rho \models \eta$.*

If the **while** loop is lossy, we have $\models \{\eta\}\ \mathbf{while}\ b\ \mathbf{do}\ s\ \{\eta\}$ if we know (i) $\models \{\eta\}\ (\mathbf{if}\ b\ \mathbf{then}\ s)^n\ \{\eta\}$ as we take the limit in n, and (ii) $\models \{\eta\}\ \ \mathbf{assert}\ \neg b\ \{\eta\}$. These considerations motivate the following definition of *downwards closedness*.

Definition 3. *An assertion η is* d-closed *if for every logical valuation ρ,*

- *for every increasing sequence $(\mu_n)_{n\in\mathbb{N}}$ of sub-distributions s.t. $\lim_{n\to\infty} \mu_n = \mu$, we have $\mu_n; \rho \models \eta$ for every $n \in \mathbb{N}$ implies $\mu; \rho \models \eta$, and*
- *for sub-distributions μ and μ' such that $\mu'(m) \leq \mu(m)$ for each $m \in \mathsf{State}$, $\mu; \rho \models \eta$ implies $\mu'; \rho \models \eta$.*

The first requirement guarantees (i): the loop invariant η will take the limit when unrolling the loop; note that this requirement only needs to considers increasing sequences of sub-distributions, since the truncated loop iterates are increasing. The second requirement guarantees (ii), since the $\mathbf{assert}$ statement only lowers the weight of the distribution.

To establish the closedness side-conditions, we use a proof system for t- and d-closedness. Both properties are closed under boolean combinations, universal quantification, and bounded existential quantification, and under logical equivalence. Therefore typical rules of the proof systems include:

$$\frac{\eta_1\ \text{tclosed} \qquad \eta_2\ \text{tclosed}}{\eta_1 \circ \eta_2\ \text{tclosed}} \qquad \frac{\eta_1\ \text{tclosed} \qquad \eta_1 \Leftrightarrow \eta_2}{\eta_2\ \text{tclosed}}$$

$$\frac{\eta\ \text{tclosed}}{\forall \dot{y}.\ \eta\ \text{tclosed}} \qquad \frac{\eta\ \text{tclosed}}{\exists \dot{y}.\psi \wedge\ \eta\ \text{tclosed}},$$

where $\circ$ ranges over $\{\wedge, \vee\}$ and ψ is satisfied by finitely many values. Analagous rules prove d-closed.

An important technical point is that closedness of atomic P-assertions may not hold in general. The rules for such atomic assertions are:

$$\frac{p_1, p_2\ \text{bounded}}{p_1 \bowtie p_2\ \ \text{tclosed}}\ \bowtie \in \{\leq, \geq, =\} \qquad \frac{p\ \text{pos-bounded}}{p \leq c\ \ \text{dclosed}},$$

where p_1, p_2 *bounded* means that they only contain integrals $\oint_\Gamma \tilde{e}$ where $\tilde{e}$ is bounded by a finite value for all substitutions of variables (program, logical, and integral). This is true, for instance, if $\tilde{e} = \mathbf{1}_\phi$ (when the integral represents the probability of a certain event). Similarly, p *pos-bounded* means p is a polynomial with non-negative coefficients of integral expressions $\oint_\Gamma \tilde{e}$ with $\tilde{e}$ non-negative and bounded above under all substitutions of variables.

To see why t-closedness may fail when the integral is not bounded, let us consider a small example. For simplicity, we consider the distribution of a single variable rather than a whole state, though the example generalizes directly. Consider a sequence of distributions $(\zeta_i)_{i\in\mathbb{N}^+}$ on $\mathbb{N}$, where

$$\zeta_i(j) = \begin{cases} 1/i & : j = i \\ 1 - 1/i & : j = 0 \\ 0 & : \text{otherwise.} \end{cases}$$

Then, $\lim_{n\to\infty} \zeta_i = \zeta^*$, where ζ^* returns 0 with probability 1. However, $\oint \zeta_i = 1$ for all i, but $\oint \zeta^* \neq 1$, violating t-closedness.

Proof rules. We tour the **while** rules loops presented in Figure 5.

The rule [WHILE-G] requires that the invariant is d-closed, but does not restrict the termination behavior of the loop. Besides [WHILE-FALSE], it is the sole rule for reasoning about non-terminating loops. However, it is also convenient for reasoning about terminating loops if the post-condition is d-closed.

The rule [WHILE-C] 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 rules [WHILE-PVB] and [WHILE-PVU] require that the loop invariant is t-closed, and enforce a.s. termination of the loop. Informally, rule [WHILE-PVB] assumes that the variant $\tilde{e}$ is (B)ounded and decreases with strictly positive probability; note that ϵ is a strictly positive real constant, not an expression. In contrast, rule [WHILE-PVU] allows an (U)nbounded variant, but it requires that variant increases by at most one at each iteration, and that it is more likely to decrease than not.

The rule [WHILE-FALSE] captures the fact that a loop whose guard is falsified by the precondition behaves like a **skip** instruction for this precondition. Similarly, the rule [WHILE-TRUE] considers loops whose guard is always true relative to a precondition η; in this case the loop has the same semantics as **abort**; this can be seen since the postcondition $\eta \wedge \Box\neg b$ entails $\Box b \wedge \Box\neg b$ and $\Box\bot$.

We conclude this section by remarking that one can generalize the rules [WHILE-PVB], [WHILE-PVU], and [WHILE-C], by considering loops invariants of the form $\eta \wedge \Box\phi$ and strengthening the precondition of the variant premise with $\Box\phi$.

5.4 Soundness

We can prove soundness of our logic; details are in the appendix.

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

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

5.5 Specialized rules

The core rules that involve branching, such as the conditional and looping rules, are powerful but somewhat heavy to use. When the guard is a deterministic expression—a common case in randomized algorithms—it is often simpler to use specialized versions of the rules. These rules are simply the normal Hoare logic rules for a deterministic language, tweaked for our setting.

To express these rules, we introduce a simple abbreviation to describe a deterministic boolean expression:

$$\circledcirc e \triangleq \Box e \vee \Box\neg e.$$

With this abbreviation, we can state deterministic rules that recover the usual Hoare rules for a deterministic language. For instance:

$$\textsc{If-D}\ \frac{\{\eta \wedge \Box e\}\ s_1\ \{\eta'\} \qquad \{\eta \wedge \Box\neg e\}\ s_2\ \{\eta'\}}{\{\eta \wedge \circledcirc e\}\ \mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2\ \{\eta'\}}$$

This rule allows reasoning about the branches separately; there is no need to reason about splitting the probabilistic state as in rule [IF], since the control flow is deterministic. This rule is derivable from the core system. We can also give a deterministic rule for loops:

$$\textsc{While-D}\ \frac{\{\eta \wedge \Box b\}\ s\ \{\eta \wedge \circledcirc b\}}{\{\eta \wedge \circledcirc b\}\ \mathbf{while}\ b\ \mathbf{do}\ s\ \{\eta \wedge \Box\neg b\}}$$

Since the guard may be modified by the loop body, we require that the guard must remain deterministic after executing the loop body, assuming the guard was initially true (deterministically). This rule is derivable from rule [WHILE-C] if we can find a deterministic invariant; otherwise, its soundness must be proved separately.

Computing preconditions is another standard tool for simplifying (and automating) proofs. Given a statement s and an assertion η, a precondition calculus computes an assertion $\eta^\star$ such that $\{\eta^\star\}\ s\ \{\eta\}$ is a valid statement. It is possible to define a precondition calculus for loop-free statements, similar to lines of Chadha et al. [8]. Importantly, our calculus, denoted by pc as usual, is defined by induction on the *assertion*, then on the *statement*, to handle the peculiar form of the rule [IF]. The full definition of pc is given in the Appendix. Our derivations freely use the rule:

$$\textsc{Precond}\ \frac{}{\{\mathrm{pc}(s,\eta)\}\ s\ \{\eta\}}.$$

6. Distribution laws and independence

The core logic features expressive assertions, but they are also verbose and can be cryptic. In this section, we define derived assertions and derived rules for higher-level properties that are ubiquitous in proofs about randomized algorithms. The derived assertions enable compact assertions, crucial when working with complex invariants. Furthermore, these properties smooth the interface with theorems from probability theory, which are often stated in terms of higher-level properties. Since the assertions are defined, we can always unfold their definitions when necessary.

We will use the following notation for expected value:

$$\mathbb{E}[\tilde{e}] \triangleq \oint \tilde{e}.$$

6.1 Distribution assertions.

When analyzing a random draw from a distribution, a fundamental piece of information is the *shape* of the distribution—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. While we can already track this information simply by recording the probability of each possible outcome of a random variable (using a universal quantifier for variables that have infinite support), such an encoding is verbose, cumbersome to use, and obscures the shape of the distribution. To alleviate this issue, we define *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:

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

We can also represent distributions with infinite support, like the geometric distribution with parameter w:

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

GENERIC RULE:

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

SIDE CONDITION: $\tilde{e} : \mathbb{N}$

$$\mathfrak{C}_{\mathrm{G}} \triangleq \eta \text{ dclosed} \qquad \mathfrak{C}_{\textsc{False}} \triangleq \eta \Rightarrow \Box\neg b \qquad \mathfrak{C}_{\textsc{True}} \triangleq \eta \Rightarrow \Box b$$

$$\begin{aligned}
\mathfrak{C}_{\mathrm{C}} \triangleq\ & \{\mathcal{L} \wedge \Box(\tilde{e} = k \wedge 0 < k \wedge b)\}\ s\ \{\mathcal{L} \wedge \Box(\tilde{e} < k)\} \\
& \models \eta \Rightarrow (\exists \dot{y}.\ \Box\tilde{e} \leq \dot{y} \wedge \Box(\tilde{e} = 0 \Rightarrow \neg b)) \\
\mathfrak{C}_{\mathrm{PVB}} \triangleq\ & \{\mathcal{L} \wedge \Box(\tilde{e} = k \wedge 0 < k \leq K \wedge b)\}\ s\ \{\mathcal{L} \wedge \Box(0 \leq \tilde{e} \leq K) \wedge \Pr[\tilde{e} < k] \geq \epsilon\} \\
& \models \eta \Rightarrow \Box(0 \leq \tilde{e} \leq K \wedge \tilde{e} = 0 \Rightarrow \neg b) \\
& \models \eta \text{ tclosed} \\
\mathfrak{C}_{\mathrm{PVU}} \triangleq\ & \{\mathcal{L} \wedge \Box(\tilde{e} = k \wedge 0 < k \wedge b)\}\ s\ \{\mathcal{L} \wedge \Box(\tilde{e} \leq k+1) \wedge \Pr[\tilde{e} < k] \geq \Pr[\tilde{e} \geq k]\} \\
& \models \eta \Rightarrow \Box(\tilde{e} = 0 \Rightarrow \neg b) \\
& \models \eta \text{ tclosed}
\end{aligned}$$

Figure 5. Core rules: loops

A natural place to introduce shape assertions is when sampling from a distribution. We have the following derived rule:

$$\textsc{Shape}\ \frac{}{\{\mathcal{L}\}\ x \xleftarrow{\$} g\ \{x \sim g\}}.$$

Shape assertions also provide a convenient hook where we can interface with facts about the distribution. For instance, we can encode facts about the range:

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

and the expectation and variance:

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

for $c \in (0,1)$ a constant. These assertions can be further manipulated via facts about expected value, like linearity of expectation.

6.2 Independence.

A very common property when analyzing randomized algorithms is *independence of random variables*. Informally, this property states that a collection of samples are drawn from distinct sources of randomness. This is a key piece of information, even when analyzing extremely simple algorithms. For instance, if we know that two samples follow a fair coin flip distribution, the probability that both samples are heads could be $1/2$ (if both samples are actually from the same coin) to 0 (if the samples always take opposite values) and all values in between, depending on the correlation between the two samples. If the samples are *independent*, then there is no correlation and the probability of seeing two heads is exactly $1/4$.

We can directly encode the following definition of independence:

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

This is the standard definition of independence, plus a normalization depending on $\Pr[\top]$ since we are working with sub-distributions.

It is also possible to reason about independence between groups of random variables, even if the variables inside each group may not be independent. This is captured by the following binary independence assertion for independent integer variables:

$$\begin{aligned}
&\langle x_1, \ldots, x_n\rangle \,\#\, \langle y_1, \ldots, y_m\rangle \triangleq \forall a_1, \ldots, a_n, b_1, \ldots, b_m \in \mathbb{Z}. \\
&\Pr[\top] \Pr[x_1 = a_1 \wedge \ldots x_n = a_n \wedge y_1 = b_1 \wedge \cdots \wedge y_m = b_m] \\
&= \Pr[x_1 = a_1 \wedge \cdots \wedge x_n = a_n] \Pr[y_1 = b_1 \wedge \cdots \wedge y_m = b_m].
\end{aligned}$$

By changing the domain of the quantifiers, we can also define independence for other types of variables, or even heterogeneous collections of variables.

While independence can be proved by expanding the abbreviation and applying the core rules, it is more convenient to apply specialized introduction rules. The first derived rule models the intuition behind independence: if some variables $\overline{x}$ are independent, then a new sample is mutually independent from $\overline{x}$.

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

We also have a more general version of this rule when the distribution may depend on random parameters:

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

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

Like the shape assertions, we can provide facts to manipulate independence symbolically. For instance, we can permute and project independence assertions:

$$\langle \overline{x}\rangle \,\#\, \langle \overline{y}\rangle \Rightarrow \langle \overline{y}\rangle \,\#\, \langle \overline{x}\rangle, \qquad \overline{x'} \subset \overline{x} \wedge \#\langle \overline{x}\rangle \Rightarrow \#\langle \overline{x'}\rangle.$$

These facts enable symbolic reasoning about independence without descending to the level of probabilities.

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 in some sense if we take many samples.

There are many forms of concentration bounds, depending on the distribution of the samples, the precise way the samples are combined, and other factors. Here, we state the *Hoeffding bound*, a basic concentration bound that is already useful to prove non-trivial facts about randomized algorithms. For constants $q, \beta \in \mathbb{R}$,

$$\begin{aligned}
&\#\langle \overline{x}\rangle \wedge \forall_{i \in [1,N]}.\ \Box(0 \leq x_i \leq 1) \wedge \mathbb{E}[x_i] \leq q \\
&\quad \Rightarrow \Pr\left[\left|\textstyle\sum_{i=1}^n x_i - n \cdot q\right| > T(\beta)\right] < \beta,
\end{aligned}$$

for $T(\beta) = 1/2 \cdot \sqrt{n \log(2/\beta)}$.

To read this fact, we first require three things: (i) the random samples $\overline{x}$ should be independent, (ii) each random sample is

bounded within $[0, 1]$, and (iii) the expected value of each random variable is bounded by q. Then, the conclusion states that with high probability (at least $1 - \beta$), the deviation from the mean from summing n random samples is on the order of $\sqrt{n}$. This is significantly smaller than the worst case, when the deviation may be on the order of n.

7. Implementation

We have built a prototype implementation called Ellora on top of EasyCrypt [4, 6], a tool-assisted framework for cryptographic proofs. We outline some of the key innovations of Ellora, and briefly discuss its relation to EasyCrypt.

First-order treatment of memories. EasyCrypt refers to program states as memories, and uses a special type **Mem** to denote memories. While the notion of memory is central to EasyCrypt, the type **Mem** does not have a first-class status: any definition or construct that involves the type **Mem** must be added by extending the EasyCrypt kernel. This expands the trusted computing base of EasyCrypt, and makes the system harder to understand and to extend. Ellora allows the definition of operators and predicates that depend on memories, making the type of memories no more special than the type of booleans. This transversal change enables formalization of distributions, probabilistic states, assertions, *etc*, from first principles within the ambient logic of Ellora.

New libraries. In order to support all assertions of the program logic, Ellora provides new libraries for big operators, subdistributions, *etc*. An important aspect of our libraries is that they are in a foundational style, i.e. they construct realizations of the objects of study instead of giving axiomatizations. A large part of our libraries are proved formally from first principles. However, some results, such as concentration bounds, are currently declared as axioms.

Our formalization of probabilistic independence deserves special mention. We formalized two different (but logically equivalent) notions of independence. The first is in terms of products of probabilities, generalizing the definition from § 6 to work with heterogenous lists of variables. Since Ellora (like EasyCrypt) has no support for heterogeneous lists, we use a smart encoding based on second-order predicates. The second definition is more abstract, in terms of product and marginal distributions. While the first definition is easier to use when reasoning about randomized algorithms, the second definition is more suited to proving mathematical facts like permutation and projection. We prove the two definitions equivalent, and formalize a collection of related theorems.

Integrating EasyCrypt and Ellora. Although EasyCrypt and Ellora are closely related, they are not fully integrated at this time. There are obvious benefits to their integration: many cryptographic proofs have conditional equivalence ("up to bad") steps, where we want to bound the probability of events—this can be done with Ellora. Conversely, many proofs of randomized algorithm do not follow the original program code closely, and would be easier to prove on a functionally equivalent program that more closely follows the line of reasoning—EasyCrypt was designed to prove such equivalences. More generally, many examples may benefit from combined relational and non-relational reasoning.

Perspective: towards a foundational system. More fundamentally, the design and implementation of Ellora marks a milestone towards building a foundational system for reasoning about probabilistic programs. Our ultimate goal is to achieve a clear separation between the underlying proof assistant and code for program logics. To this end, we still need to enable a deep embedding of probabilistic programs, by giving statements a first-class treatment and formalizing their semantics. In this way, it will be possible to prove the rules of our program logic (and the relational program logic of EasyCrypt) from first principles. Besides reducing the trusted computing base significantly, this redesign would allow users to add new rules or to build certified program logics, perhaps for the pWhile language (e.g., [5]), but also conceivably for other languages [7]. Combined with fully foundational libraries, we envision an easily extensible system where the trusted computing base consists exclusively of a simple and readable checker for a lightweight higher-order logic.

8. Examples

In this section, we will demonstrate Ellora on a selection of examples. Some of the examples are drawn from algorithms textbooks, while other come from the algorithms research literature. Together, they exhibit a wide variety of different proof techniques and reasoning principles, while demonstrating various uses of randomization in algorithmic design.

8.1 Randomization for approximation: vertex cover

We begin with a classical application of randomization: *approximation algorithms* for computationally hard problems. The idea is that even if a particular problem takes a long time to solve in the worst case, we can devise efficient algorithms that return a solution that is almost as good as the true solution, for some definition of "almost". While there is certainly no requirement that approximation algorithms must be randomized, in practice the simplest and best-performing approximation algorithms often use randomization.

Our first example illustrates a famous approximation algorithm for the *vertex cover* problem. In this 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, so it is unlikely that there is an efficient algorithm for computing the optimal vertex cover. However, there is simple randomized algorithm that returns a vertex cover that is at most twice the size of the optimal vertex cover, on average. The algorithm proceeds by considering each edge in order. If neither endpoint is in the cover, the algorithm chooses one of the two endpoints uniformly at random. In Ellora, this corresponds to the following program:

```
param set<edge> E;
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.

To verify the invariant, clearly it 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 $\mathtt{e}$ is not covered in $\mathtt{C}$ 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

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

and by reasoning on intersection and difference of sets, we have

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

In particular, if $\mathtt{C}^*$ is the optimal vertex cover, this judgment shows that our vertex cover is at most twice as large as optimal.

8.2 Random walks: termination and reachability

Next, we turn to a different application of randomization: describing random processes, where a random quantity evolves over time. A canonical example is a *random walk*; there are many variations, but the basic scenario describes a numeric position changing over time, where the position depends on the position at the previous timestep, influenced by random noise drawn from some distribution.

To demonstrate how to verify interesting facts about random processes, we will model a one-dimensional random walk on the natural numbers. We start at position 0, and repeatedly update our position according to the following rules. From 0, we always make a step to 1. From non-zero positions, we flip a fair coin that is biased to come up heads with probability $p > 0$. If heads, we increase our position by 1; if tails, we decrease by 1.

We will prove two facts about this random walk. First, for any natural number T, the probability of eventually reaching T is 1. Second, when we reach T, we must first pass through all intermediate points $1, \ldots, T-1$. In Ellora, we can express the random walk with the following code.

```
var bool visited[T];
var int  pos = 0;
visited[0] ← true;
while pos ≠ T do:
  c  ←$ Ber(p);
  pos ← pos + ((pos = 0) ∨ c) ? 1 : -1;
  visited[pos] ← true;
end
```

In order to verify this example, we will use the probabilistic while rule [WHILE-PVB]. First we establish almost-sure termination by finding an appropriate termination measure: the distance between our current position, and T. Indeed, this measure is bounded by T, and has a non-negative (at least $1/2$) probability of decreasing each iteration. Therefore, the loop terminates almost-surely, and thus our random walk eventually reaches any point T with probability 1.

To prove our second assertion—that we visit every point from 0 to T—we use the following loop invariant for the while command:

$$\Box(\forall i.\ 0 \leq i \leq \mathtt{pos} \Rightarrow \mathtt{visited[i]} = \mathtt{true}).$$

In other words, if we have reached position pos, then we must have already reached every position in $[0, \mathtt{pos}]$. Since this invariant is t-closed, we may apply rule [WHILE-PVB] and the invariant holds at the end of the loop. With the assertion $\mathtt{pos} = \mathtt{T}$ at termination, we have enough to prove that each position is visited:

$$\{\mathcal{L}\}\ s\ \{\mathcal{L} \wedge \Box(\forall i \in [\mathtt{T}].\ \mathtt{visited[i]} = \mathtt{true})\}.$$

The losslessness post-condition indicates that the walk terminates almost-surely.[3]

[3] We remark that an unbounded version of the random walk, where the walk may decrease from position 0, can be proved terminating by rule [WHILE-PVU].

8.3 Amplification: Polynomial identity testing

A second use of randomness is in running independent trials of the same algorithm. This technique, known as *probability amplification*, runs a randomized algorithm several times in order to reduce the error probability. Roughly speaking, a single trial may have high error with some probability, but by repeating the trial it is unlikely that all of the trials yield poor results. By combining the results appropriately—e.g., with a majority vote for algorithms with binary outputs, or by selecting the best answer when we can check the quality of the solution—we can produce an output that is more accurate than a single run of the original algorithm.

An example in this vein is probabilistic *polynomial identity testing*. Given two multivariate polynomials $P(x_1, \ldots, x_n)$ and $Q(x_1, \ldots, x_n)$ over the finite field $\mathbb{F}_q$ of q elements, we want to check whether $P = Q$, or equivalently, whether the polynomial $P - Q$ is zero or not. We will take n uniformly random samples $(v_i)_i$ from $\mathbb{F}_q$ and check whether $(P-Q)(v_1, \ldots, v_n) = 0$. We repeat the trial q times, rejecting if we see a sample where the difference is non-zero. In our system, this corresponds to the following program:

```
var bool res = true;
for i = 1 to q do:
  v   ←$ Unif(F_q^n);
  res ← res ∧ ((P - Q)(v) = 0);
end
```

The proof uses an instance of the Schwartz-Zippel lemma due to Øystein Ore for finite fields, which upper bounds the probability of randomly picking a root of a polynomial over a finite field.

Lemma 1. *Let P a non-zero polynomial function over $\mathbb{F}_q$. If we sample the variables $v_1, \ldots, v_n$ uniformly at random from $\mathbb{F}_q$, then*

$$\Pr[P(v_1, \ldots, v_n) = 0] \leq 1 - 1/q.$$

We encode this lemma as an axiom in our system:

$$P \neq 0 \wedge v \sim \mathbf{Unif}(\mathbb{F}_q^n) \Rightarrow \Pr[P(v) = 0] \leq 1 - 1/q.$$

With this fact, we can prove the following loop invariant:

$$\mathtt{P} \neq \mathtt{Q} \Rightarrow \Pr[\mathtt{res} = \mathtt{true}] \leq (1 - 1/\mathtt{q})^i,$$

finally proving that

$$\{\mathtt{P} \neq \mathtt{Q}\}\ s\ \{\Pr[\mathtt{res} = \mathtt{true}] \leq (1 - 1/\mathtt{q})^{\mathtt{q}} \leq 1/e\}.$$

We have also verified Freivald's algorithm, an amplification-based algorithm for checking matrix multiplication.

8.4 Modeling infinite processes: the coupon collector

Now, we will revisit the coupon collector algorithm from § 2:

```
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 involves two nested loops, and so we have two loop invariants. The loop invariant for the outer loop is relatively standard,

since the loop has a fixed bound $\mathtt{N}$ on the number of iterations:

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

where $\rho(i) \triangleq (\mathtt{N}-i+1)/\mathtt{N}$. We can apply the while rule [WHILE-C] to verify the outer loop.

The first conjunction states that the previous waiting times are distributed according to a geometric distribution with parameter that depends on i. Intuitively, this is because as we collect more coupons, we are less likely to see a new one. 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 turned on in $\mathtt{cp}$.

Handling the inner while loop is more complicated, since it has a probabilistic guard. To select the appropriate while rule, we consider the termination behavior of the inner loop. 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 portion of [WHILE-PVB] is verified.

For the main loop invariant, we use the following formula:

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

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 about this loop invariant, for every natural number c there are two cases. Either we have already executed more than c iterations, or not. In the first case, we have the first disjunction: loops where the guard is true all have $\mathtt{ct} \geq c$, and the total probability of stopping at exactly c iterations is $(1-\rho(\mathtt{p}))^c\rho(\mathtt{p})$—we see c old coupons, then we see a new one. In the second case, we have the second disjunction. There exists an integer k that represents the current iteration; if the loop is continuing then $k = \mathtt{ct}$, and the loop stops in the future ($k \leq \mathtt{ct}$). Furthermore, the probability of continuing at iteration k is $(1-\rho(\mathtt{p}))^k$—we have see k old coupons.

To conclude, we know that at the end of the loop $\Box\mathtt{cp[cur]} = 0$. 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 $\mathtt{ct}$ as $\mathbf{Geom}(\rho(\mathtt{p}))$ by definition.

The outer loop is easier to handle, since the loop has a fixed bound $\mathtt{N}$ 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 $\mathtt{cp}$. Putting it all together, we have

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

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

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

8.5 Concentration bounds: private running sums

Now, we turn to examples involving independence of random variables. Our first example is drawn from the *differential privacy* literature. Given a list of N integers, we add noise from a two-sided geometric distribution to each entry in order to protect privacy, and we calculate the *partial sums* of the noisy values: $x_1, x_1+x_2, x_1+x_2+x_3$, and so on. We wish to measure how far the noisy sums deviate from the true sums.

In Ellora, we can express this algorithm as follows:

```
var int X[N], noise[N], out[N];
var int acc = 0;
for i = 1 to N do
  noise[i] ←$ twogeom(ϵ);
  out[i]   ← acc + X[i] + noise[i];
  acc      ← out[i];
end
```

The parameter ϵ to the noise distribution `twogeom` is a numeric parameter controlling the strength of the privacy guarantee, by changing the magnitude of the noise.

Our loop invariant tracks three pieces of information: (i) `noise[i]` is distributed according to $\mathtt{twogeom}(\epsilon)$; (ii) the array `noise` remains independent; and (iii) `out[i]` stores the noisy running sum:

$$\forall q \in [\mathtt{i}].\ \mathtt{out[q]} - \sum_{p\in[q]} \mathtt{X}[p] = \sum_{p\in[q]} \mathtt{noise}[p].$$

To bound the error introduced by the noise, we need to bound $\left|\sum_{p\in[q]} \mathtt{noise}[p]\right|$. Since we know that the elements in `noise` are all independent, we can apply a concentration bound to bound the probability of a large error in the p-th running sum, concluding:

$$\{\mathcal{L}\}\ s\ \left\{\forall p \in [\mathtt{N}].\ \Pr\left[\left|\sum_{i\in[p]} \mathtt{X}_i\right| \geq T\right] \leq Q(\epsilon, T)/\sqrt{\mathtt{N}}\right\}$$

for a particular function Q derived from the Berry-Esseen theorem.

8.6 Conserving randomness: pairwise-independent bits

As we have seen in the previous examples, independent random bits are a powerful tool in randomized algorithms. However, they are also scarce resource—fresh randomness for each bit is needed to ensure independence. For many applications, e.g. hashing, we can get by with a weaker notion of independence: *pairwise independence*.

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 many of pairwise-independent bits. Here, we consider one procedure:

```
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
```

Here, $\oplus$ represents the boolean XOR operation. The 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-valued functions f, g. To complete the verification, note that

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

where the big XOR operator ranges over the indices j where the bit representation of i has bit j set. For any two $i, k \in [1, \ldots, 2^{\mathsf{N}}]$ distinct, there is a bit position in $[1, \ldots, \mathsf{N}]$ where i and k do not agree. Without loss of generality, call this position r and suppose it is set in i but not in k. By rewriting,

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

Since $\mathsf{B}[j]$ are all independent, $\mathsf{X}[i] \mathbin{\#} \mathsf{X}[k]$ follows from eq. (2) taking Z to be the distribution on tuples $\langle \mathsf{B}[1], \ldots, \mathsf{B}[\mathsf{N}] \rangle$ excluding $\mathsf{B}[r]$. This verifies the claimed specification:

$$\{\mathcal{L}\}\ s\ \{\mathcal{L} \wedge \forall i, k \in [2^{\mathsf{N}}].\ i \neq k \Rightarrow \mathsf{X}[i] \mathbin{\#} \mathsf{X}[k]\}.$$

9. Related work

Verification of probabilistic programs. There is a long tradition of research in the formal verification of probabilistic programs. Early works by Feldman and Harel [14], Kozen [27], Ramshaw [36], Reif [38], Sharir et al. [41] propose formalisms for proving properties of probabilistic programs, and use them for verifying several examples of interest. For instance, Feldman and Harel [14] prove the correctness of a program computing the geometric distribution, whereas Kozen [27], Sharir et al. [41] prove termination properties of the one-dimensional random walk. In a series of works initiated by Morgan et al. [34] and described in their textbook [31], McIver, Morgan, and their collaborators develop deductive verification methods for programs written in pGCL, an imperative language with probabilistic choice and (demonic) non-determinism. In their work, assertions are measures over distributions, and the semantics of programs are based on weakest preconditions. Hurd et al. [23] formalize pGCL in the HOL4 theorem prover, and verify key parts of Rabin's mutual exclusion algorithm. Katoen et al. [26] develop an automated method to infer linear probabilistic invariants for pGCL programs; later, Gretz et al. [18] improve and implement their method. Recently, Gretz et al. [19] extend pGCL to account for conditioning.

In his thesis, den Hartog [13] develops an alternative approach based on Hoare-style rules rather than weakest preconditions. Soundness for the rule for **while** loops is achieved through a semantic closure condition. The use of semantic conditions is undesirable for verification, because it requires reasoning about the semantics of programs. More recent works develop program logics for restricted settings. Chadha et al. [8] give a decidable Hoare logic for a probabilistic language without **while** loops; decidability imposes strong restrictions on program values and limits the applicability of the logic. Rand and Zdancewic [37] 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 some strong restrictions on **while** loops. Similarly, EasyCrypt implements an (unpublished) logic to reason about probability of events, but it restricts the class of properties and relies on a sequence rule that is difficult to use.

There has been significant work to develop and apply static or dynamic techniques for inferring properties of probabilistic programs. In a series of works initiated by Monniaux [33], Monniaux develops an abstract interpretation framework for probabilistic programs. Recent work by Cousot and Monerau [12] proposes a more general framework. An elegant method based on martingales is used by Chakarov and Sankaranarayanan [9, 10] for inferring expectation invariants and other properties. Using their method, they estimate 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 [15] for analyzing probablistic termination. Sampson et al. [40] use a mix of static and dynamic analysis to check properties of probabilistic algorithms from the approximate computing and privacy literature.

Other relevant work include the development of Markovian logics [29], coinductive reasoning principles for stochastic processes [28], Hoare logics for quantum programs [42], relational Hoare logics for probabilistic programs [3, 5, 7], *etc*. Furthermore, there is significant work in model-checking (see e.g., Katoen [25]).

Machine-checked proofs of probability theory. Formalizations of measure and integration theory were proposed by Coble [11], Hölzl and Heller [20], Hurd [22], Mhamdi et al. [32], Richter [39]. These works do not formalize concentration bounds. However, Avigad et al. [2] recently completed a proof of the Central Limit Theorem, which is the principle underlying concentration bounds. Audebaud and Paulin-Mohring [1] propose an alternative, more axiomatic, approach for discrete distributions, and use it for building a library for reasoning about functional probabilistic programs. These formalizations have been used to verify several case studies; for instance, Hurd [22] verifies the Miller-Rabin primality test, and Hölzl and Nipkow [21] verify the Crowds protocol.

10. Conclusion

We have developed and implemented a general verification platform for randomized programs, and shown the feasibility of proving non-trivial examples. On the one hand, we view our work as merely a first step, with many directions for extending the power of our platform; for example, it may be possible to incorporate tools from martingale theory to provide a finer analysis of almost-surely terminating loops.

On the other hand, we believe our system is already powerful enough to contemplate formalization in many areas of theoretical computer science. Prime targets include proving accuracy and differential privacy of algorithms; reductions from complexity theory and lower bounds; and distributed algorithms. Our techniques could also be applied to more mathematical areas, like combinatorics proofs based on the probabilistic method.

Finally, further integration with relational systems like EasyCrypt could bring many benefits, notably the possibility to combine relational and non-relational reasoning. This could be useful for verifying proofs of randomized algorithms that rely on techniques like probabilistic coupling. More broadly, our implementation gives hope for building a truly foundational interactive theorem prover for randomized programs.

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] G. Barthe, B. Grégoire, and S. Z. Béguelin. Formal certification of code-based cryptographic proofs. In Z. Shao and B. C. Pierce, editors, *Proceedings of the 36th ACM Symposium on Principles of Programming Languages, POPL 2009*, pages 90–101. ACM, 2009.

[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, B. Köpf, F. Olmedo, and S. Z. Béguelin. Probabilistic relational reasoning for differential privacy. In J. Field and M. Hicks, editors, *Proceedings of the 39th ACM Symposium on Principles of Programming Languages, POPL 2012*, pages 97–110. ACM, 2012.

[6] 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.

[7] G. Barthe, C. Fournet, B. Grégoire, P. Strub, N. Swamy, and S. Z. Béguelin. Probabilistic relational verification for cryptographic implementations. In S. Jagannathan and P. Sewell, editors, *41st ACM Symposium on Principles of Programming Languages, POPL '14*, pages 193–206. ACM, 2014.

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

[9] 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.

[10] 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.

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

[12] 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.

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

[14] Y. A. Feldman and D. Harel. A probabilistic dynamic logic. *J. Comput. Syst. Sci.*, 28(2):193–215, 1984.

[15] 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.

[16] C. Gentry, C. Peikert, and V. Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In C. Dwork, editor, *Proceedings of the 40th ACM Symposium on Theory of Computing, 2008*, pages 197–206. ACM, 2008.

[17] A. D. Gordon, T. A. Henzinger, A. V. Nori, and S. K. Rajamani. Probabilistic programming. In *International Conference on Software Engineering*, pages 167–181, 2014.

[18] 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.

[19] 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.

[20] 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.

[21] J. Hölzl and T. Nipkow. Interactive verification of markov chains: Two distributed protocol case studies. In U. Fahrenberg, A. Legay, and C. R. Thrane, editors, *Quantities in Formal Methods, QFM 2012*, volume 103 of *EPTCS*, pages 17–31, 2012.

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

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

[24] A. Joux. A new index calculus algorithm with complexity l(1/4+o(1)) in small characteristic. In *Selected Areas in Cryptography - SAC 2013*, pages 355–379, 2013.

[25] 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.

[26] 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.

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

[28] D. Kozen. Coinductive proof principles for stochastic processes. *Logical Methods in Computer Science*, 3(4:8), 2007.

[29] D. Kozen, R. Mardare, and P. Panangaden. Strong completeness for Markovian logics. In K. Chatterjee and J. Sgall, editors, *Mathematical Foundations of Computer Science (MFCS 2013)*, volume 8087 of *Lect. Notes in Computer Science*, pages 655–666. Springer, 2013.

[30] D. A. Levin and Y. Peres. Pólya's theorem on random walks via pólya's urn. *The American Mathematical Monthly*, 117(3):220–231, 2010.

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

[32] 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.

[33] 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.

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

[35] J. R. Norris. *Markov chains*. Number 2008. Cambridge university press, 1998.

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

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

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

[39] 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.

[40] 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.

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

[42] M. Ying. Floyd-hoare logic for quantum programs. *ACM Trans. Program. Lang. Syst.*, 33(6):19, 2011.

A. Soundness

The proof of soundness of the presented proof system relies on the soundness of each rule. We detail here the proofs for each case.

A.1 Deterministic assignment

The soundness of the deterministic assignment rule follows from similar reasoning as in Hoare logic. Roughly speaking, when the formula ϕ interpreted in a state after the assignment of a variable x is equivalent to the replacement of all the occurrences of x by its new value in ϕ. This intuition is captured in the following Lemma:

Lemma 2 (Substitution lemma for state and distribution expressions)**.** *For a given state $m \in \mathsf{State}$, a valuation of the logical variables ρ, a variable of the language x, and a state expression $\tilde{e}$ we have, for all constants e:*

$$[\![\tilde{e}[x := e]]\!]^{\rho}_{m} = [\![\tilde{e}]\!]^{\rho}_{m[x:=[\![e]\!]_m]},$$

and similarly, for all distribution expressions g, we have:

$$[\![g[x := e]]\!]^{\rho}_{m} = [\![g]\!]^{\rho}_{m[x:=[\![e]\!]_m]},$$

Proof. By mutual induction on the structure of the expression $\tilde{e}$ and g. □

We can now lift this first lemma to the expressions interpreted in a probabilistic state.

Lemma 3 (Substitution lemma for integral expressions)**.** *For a given sub-distribution μ, a valuation ρ, a variable of the language x, and a state expression $\tilde{e}$ and a program expression e, we have*

$$[\![\oint_{\Gamma[x:=e]} \tilde{e}[x := e]]\!]^{\rho}_{\mu} = [\![\oint_{\Gamma} \tilde{e}]\!]^{\rho}_{[\![x:=e]\!]_\mu}.$$

Proof. We will prove the case for a single $\Gamma = (t, g)$; the general case is essentially the same. By definition we have:

$$[\![\oint_{(t,g[x:=e])} \tilde{e}[x := e]]\!]^{\rho}_{\mu} = \sum_m \sum_t [\![\tilde{e}[x := e]]\!]^{\rho}_{\mu}\ [\![g[x := e]]\!]^{\rho}_{\mu}(t)\mu(m).$$

Lemma 2 ensures that we have:

$$[\![\tilde{e}[x := e]]\!]^{\rho}_{m} = [\![\tilde{e}]\!]^{\rho}_{m[x:=[\![e]\!]^{\rho}_m]}$$

and

$$[\![g[x := e]]\!]^{\rho}_{m} = [\![g]\!]^{\rho}_{m[x:=[\![e]\!]^{\rho}_m]},$$

so

$$\sum_m \sum_t [\![\tilde{e}[x := e]]\!]^{\rho}_{\mu}\ [\![g[x := e]]\!]^{\rho}_{\mu}(t)\mu(m) = \sum_m \sum_t [\![\tilde{e}]\!]^{\rho}_{m[x:=[\![e]\!]^{\rho}_m]}\ [\![g]\!]^{\rho}_{m[x:=[\![e]\!]^{\rho}_m]}(t)\mu(m).$$

By the variable replacement $b \triangleq m \mapsto m[x := [\![e]\!]_m]$, we obtain

$$\sum_m \sum_t [\![\tilde{e}]\!]^{\rho}_{m[x:=[\![e]\!]^{\rho}_m]}\ [\![g]\!]^{\rho}_{m[x:=[\![e]\!]^{\rho}_m]}(t)\mu(m) = \sum_m \sum_t [\![\tilde{e}]\!]^{\rho}_{m}\ [\![g]\!]^{\rho}_{m}(t)(\mu \circ b^{-1})(m).$$

Now for all $A \subset \mathsf{State}$ we have by definition of the inverse image

$$\mu \circ b^{-1}(A) = \sum_m \mathbf{1}_{m[x:=[\![e]\!]_m] \in A}\mu(m) = [\![x := e]\!]_\mu(A),$$

so we have:

$$[\![\oint_{(t,g[x:=e])} \tilde{e}[x := e]]\!]^{\rho}_{\mu} = \sum_m \sum_t [\![\tilde{e}]\!]^{\rho}_{m}\ [\![g]\!]^{\rho}_{m}(t)([\![x := e]\!]_\mu)(m) = [\![\oint_{(t,g)} \tilde{e}]\!]^{\rho}_{[\![x:=e]\!]_\mu}.$$

as desired. □

Proposition 1 (Soundness of rule [ASSGN])**.** *Given a formula η and a variable x from the programming language, then for every interpretation ρ, and every sub-measure μ such that $\mu; \rho \models \eta[x := e]$ then*

$$[\![x := e]\!]_\mu; \rho \models \eta.$$

Proof. We prove something stronger: for every variable x, logical valuation ρ, sub-distribution μ, and probability expression p, we have

$$[\![p[x := e]]\!]^{\rho}_{\mu} = [\![p]\!]^{\rho}_{[\![x:=e]\!]_\mu},$$

and for every probability assertion η, we have

$$[\![\eta[x := e]]\!]^{\rho}_{\mu} = [\![\eta]\!]^{\rho}_{[\![x:=e]\!]_\mu}.$$

Done by induction first over the structure of p using Lemma 3 for integral expressions, and then over η. □

We now treat the case of the probabilistic assignment. From now on, we consider a probability distribution $\mathcal{D}(e)$ that depends on an expression e.

A.2 Probabilistic assignment

Lemma 4 (Substitution for probabilistic assignment)**.** *For any sub-distribution μ, valuation ρ, and state expression $\tilde{e}$, we have*

$$[\![\oint_{\Gamma} \tilde{e}]\!]^{\rho}_{[\![x \xleftarrow{\$} \mathcal{D}(e)]\!]_{\mu}} = [\![\oint_{(v,\mathcal{D}(e))::\Gamma[x:=v]} \tilde{e}[x := v]]\!]^{\rho}_{\mu}$$

if both sides exist.

Proof. For simplicity of notation we prove just the case when there is a single integral variable t and distribution expression g. By unfolding the semantics, we have

$$[\![x \xleftarrow{\$} \mathcal{D}(e)]\!]_{\mu}(m) = \sum_{m'} \sum_{v} \mathbf{1}_{m'[x:=v]=m} [\![\mathcal{D}(e)]\!]^{\rho}_{m'}(v)\mu(m').$$

Also, we have

$$\begin{aligned}[\![\oint_{\Gamma} \tilde{e}]\!]^{\rho}_{[\![x \xleftarrow{\$} \mathcal{D}(e)]\!]_{\mu}} &= \sum_{m} \sum_{t} \sum_{m'} \sum_{v} [\![\tilde{e}]\!]^{\rho}_{m} \mathbf{1}_{m'[x:=v]=m} [\![g]\!]^{\rho}_{m}(t) [\![\mathcal{D}(e)]\!]^{\rho}_{m'}(v)\mu(m') \\ &= \sum_{m'} \sum_{t} \sum_{v} [\![\tilde{e}]\!]^{\rho}_{m'[x:=v]} [\![\mathcal{D}(e)]\!]^{\rho}_{m'}(v) [\![g]\!]^{\rho}_{m'[x:=v]}(t)\mu(m').\end{aligned}$$

By Lemma 2, the right hand side is equal to

$$\sum_{m'} \sum_{t} \sum_{v} [\![\tilde{e}[x := v]]\!]^{\rho}_{m'} [\![g[x := v]]\!]^{\rho}_{m'}(t) [\![\mathcal{D}(e)]\!]_{m'}(v)\mu(m') = [\![\oint_{(v,\mathcal{D}(e))::g[x:=v]} \tilde{e}[x := v]]\!]^{\rho}_{\mu}$$

as desired. □

Proposition 2 (Soundness of rule [SAMPLE])**.** *Given a probability formula η, and a variable x from the programming language, then for every interpretation ρ, and every sub-measure μ such that $\mu; \rho \models \mathcal{P}^{x}_{\eta}(\mathcal{D}(e))$ then*

$$[\![x \xleftarrow{\$} \mathcal{D}(e)]\!]_{\mu}; \rho \models \eta.$$

Proof. We prove something stronger: for every variable x, logical valuation ρ, sub-distribution μ, and probability expression p, we have

$$[\![\mathcal{P}^{x}_{p}(\mathcal{D}(e))]\!]^{\rho}_{\mu} = [\![p]\!]^{\rho}_{[\![x \xleftarrow{\$} \mathcal{D}(e)]\!]_{\mu}},$$

and for every probability assertion η, we have

$$[\![\mathcal{P}^{x}_{\eta}(\mathcal{D}(e))]\!]^{\rho}_{\mu} = [\![\eta]\!]^{\rho}_{[\![x \xleftarrow{\$} \mathcal{D}(e)]\!]_{\mu}}.$$

Done by simple induction over the structure of p, using the Lemma 4 for integral expressions, and then by induction on the structure of η. □

A.3 Conditional Branching

Proposition 3 (Soundness of rule [IF])**.** *Given an expression e and two commands s_1, s_2, such that the triples $\{\eta_1\}\ s_1\ \{\eta_1'\}$ and $\{\eta_2\}\ s_2\ \{\eta_2'\}$ are valid, and μ a sub-distribution, ρ a valuation such that*

$$\mu; \rho \models (\eta_1 \wedge \Box e) \oplus (\eta_2 \wedge \Box \neg e),$$

we have

$$[\![\mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2]\!]_{\mu}; \rho \models \eta_1' \oplus \eta_2'.$$

Proof. By hypothesis and definition of $\oplus$, we have μ_1 and μ_2 such that $\mu = \mu_1 + \mu_2$ with

$$\mu_1; \rho \models \eta_1 \wedge \Box e \quad \text{and} \quad \mu_2; \rho \models \eta_2 \wedge \Box \neg e.$$

Therefore for $i = 1, 2$, if we let $\mu_i' = [\![s_i]\!]_{\mu_i}$, by definition of the semantics we have

$$[\![\mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2]\!]_{\mu} = [\![\mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2]\!]_{(\mu_1+\mu_2)} = \mu_1' + \mu_2'.$$

Moreover, by hypothesis we have $[\![\eta_i']\!]^{\rho}_{\mu_i'}$ so, by definition of the interpretation of the $\oplus$ operator, we have:

$$\mu_1' + \mu_2'; \rho \models \eta_1' \oplus \eta_2,$$

so

$$[\![\mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2]\!]_{\mu}; \rho \models \eta_1' \oplus \eta_2',$$

as desired. □

A.4 Certain termination

Proposition 4 (Soundness of rule [WHILE-C]). *For any sub-distributions, valuations μ, ρ, such that $\mathfrak{C}_{\mathbf{C}}$ is valid. Then,*

$$[\![\mathbf{while}\, b\, \mathbf{do}\, s]\!]\mu; \rho \models \eta \wedge \Box\neg b.$$

Proof. Let μ be a sub-distribution satisfying the precondition η under logical valuation ρ. By assumption η also satisfies $\exists \dot{y}.\ \Box \tilde{e} \leq \dot{y}$. Let State_n the support[4] of the distribution $\mu_n = [\![(\mathbf{if}\, b\, \mathbf{then}\, s)^n]\!]_\mu$ for $n \in \mathbb{N}$.

We first prove that there exists a decreasing function $f : \mathbb{N} \to \mathbb{N}$ such that:

$$\forall n \in \mathbb{N}, \quad \mu_n; \rho \models \Box(\tilde{e} \leq f(n) \vee \neg b).$$

By the precondition, there exists a value k such that $\mu_0; \rho \models \Box(\tilde{e} \leq k)$. Define:

$$f(n) = \begin{cases} k - n & \text{if } n \leq k \\ 0 & \text{otherwise.} \end{cases}$$

We now prove the desired assertion by induction over n. The base case $n = 0$ is clear. For the inductive step, suppose we know the result for n. By definition of $\Box$:

$$\begin{aligned}
\Box(\tilde{e} \leq f(n+1) \vee \neg b) &\equiv \oint \mathbf{1}_{\tilde{e} \leq f(n+1) \vee \neg b} = \oint 1 \\
&\equiv \oint (\mathbf{1}_{\tilde{e} \leq f(n+1)} + \mathbf{1}_{\neg b} - \mathbf{1}_{\tilde{e} \leq f(n+1)} \mathbf{1}_{\neg b}) = \oint 1 \\
&\equiv \oint \mathbf{1}_{\tilde{e} \leq f(n+1)} (1 - \mathbf{1}_{\neg b}) = \oint 1 - \oint \mathbf{1}_{\neg b} \\
&\equiv \oint \mathbf{1}_{\tilde{e} \leq f(n+1)} \mathbf{1}_b = \oint \mathbf{1}_b.
\end{aligned}$$

Since by definition of the semantics we have

$$\mu_{n+1} = [\![s]\!](\mu_{n\,|b}) + \mu_{n\,|\neg b}$$

we have by linearity and property of conditioning:

$$\begin{aligned}
\Box(\tilde{e} \leq f(n+1) \vee \neg b) &\equiv \oint \mathbf{1}_{\tilde{e} \leq f(n+1)} \mathbf{1}_b = \oint \mathbf{1}_b \\
&\equiv \sum_m \mathbf{1}_{[\![\tilde{e}]\!]^\rho_m \leq f(n+1)} \mathbf{1}_{[\![b]\!]^\rho_m} ([\![s]\!]_{\mu_{n\,|b}})(m) + \sum_m \mathbf{1}_{[\![\tilde{e}]\!]^\rho_m \leq f(n+1)} \mathbf{1}_{[\![b]\!]^\rho_m} (\mu_{n\,|\neg b})(m) \\
&\qquad = \sum_m \mathbf{1}_{[\![b]\!]^\rho_m} ([\![s]\!]_{\mu_{n\,|b}})(m) + \sum_m \mathbf{1}_{[\![b]\!]^\rho_m} (\mu_{n\,|\neg b})(m) \\
&\equiv \sum_m \mathbf{1}_{[\![\tilde{e}]\!]^\rho_m \leq f(n+1)} \mathbf{1}_{[\![b]\!]^\rho_m} ([\![s]\!]_{\mu_{n\,|b}})(m) = \sum_m \mathbf{1}_{[\![b]\!]^\rho_m} ([\![s]\!]_{\mu_{n\,|b}})(m).
\end{aligned}$$

We now use the equivalence between the $\Box$ modality and the entailment over each state. Since for every state m in the support of $\mu_{n\,|b}$ we have $m; \rho \models b$, we deduce from the induction hypothesis that $m; \rho \models \tilde{e} \leq f(n)$. Thus for every state m in the support of $[\![s]\!]_{\mu_{n\,|b}}$ we have $m; \rho \models \tilde{e} < f(n)$ i.e. $m; \rho \models \tilde{e} \leq f(n) - 1 = f(n+1)$ which implies the equality of the functions $\mathbf{1}_{\tilde{e} \leq f(n+1)} \mathbf{1}_b$ and $\mathbf{1}_b$ over the support of the considered sub-distribution, and then implies the equality when summing.

Then for all $n \geq f(0)$ we deduce that we have $\mu_n; \rho \models \Box(\tilde{e} = 0 \vee \neg b)$ and then $[\![\Box\neg b]\!]^\rho_{\mu_n}$ from the premises. We proved therefore that the loop exited after at most $f(0)$ steps, i.e. $[\![(\mathbf{if}\, b\, \mathbf{then}\, s)^{f(0)}]\!] = [\![(\mathbf{if}\, b\, \mathbf{then}\, s)^{f(0)}; \mathbf{assert}\, \neg b]\!] = [\![\mathbf{while}\, b\, \mathbf{do}\, c]\!]$. We can then conclude on the entailment of formula η by a finite induction over $\{1, \ldots, f(0)\}$. □

A.5 Almost sure termination

We will prove soundness for the rules [WHILE-PVB] and [WHILE-PVU] by first proving soundness for a more general while rule:

$$\text{WHILE-G}\ \frac{\{\eta\}\ \mathbf{if}\, b\, \mathbf{then}\, s\ \{\eta\} \qquad \text{The loop is a.s. terminating} \qquad \eta\ \text{ tclosed}}{\{\eta\}\ \mathbf{while}\, b\, \mathbf{do}\, s\ \{\eta \wedge \Box\neg b\}}$$

By the form of the side-conditions, soundness of [WHILE-PVB] and [WHILE-PVU] will follow from proving almost-sure termination. The soundness [WHILE-G] is done in the two following lemmas.

Proposition 5. *For any sub-distributions, valuations μ, ρ, such that $\{\eta\}\ \mathbf{if}\, b\, \mathbf{then}\, s; \mathbf{assert}\, \neg b\ \{\eta\}$ is verified, and η is t-closed, then*

$$[\![\mathbf{while}\, b\, \mathbf{do}\, c]\!]_\mu; \rho \models \eta.$$

Proof. The limit of $[\![(\mathbf{if}\, b\, \mathbf{then}\, c\, \mathbf{else}\, \mathbf{skip})^n; \mathbf{assert}\, \neg b]\!]_\mu$ when $n \to \infty$ exists and is $[\![\mathbf{while}\, b\, \mathbf{do}\, s]\!]_\mu$ by definition. A simple induction on the variable n implies that:

$$\forall n \in \mathbb{N}.\ [\![(\mathbf{if}\, b\, \mathbf{then}\, c\, \mathbf{else}\, \mathbf{skip})^n; \mathbf{assert}\, \neg b]\!]_\mu; \rho \models \eta$$

[4] Set of states of non-zero measure

and thus that for all natural integers n the probabilistic state $[\![(\mathbf{if}\ b\ \mathbf{then}\ c\ \mathbf{else\ skip})^n; \mathbf{assert}\ \neg b]\!]_\mu$ is included in the set $(\eta^\rho_{(-)})^{-1}(\top)$. By topological closeness of this set, the limit of this sequence remains an element of it, i.e.,

$$[\![\mathbf{while}\ b\ \mathbf{do}\ s]\!]_\mu; \rho \models \eta$$

as desired. □

Proposition 6 (Soundness of rule [WHILE-G])**.** *For any sub-distributions, valuations μ, ρ, such that $\{\eta\}\ \mathbf{if}\ b\ \mathbf{then}\ s\ \{\eta\}$ is valid, η is t-closed, and that the loop $\mathbf{while}\ b\ \mathbf{do}\ s$ terminates with probability 1, then*

$$[\![\mathbf{while}\ b\ \mathbf{do}\ s]\!]_\mu; \rho \models \eta \wedge \Box\neg b.$$

Proof. We take a sub-distribution and valuation ρ, μ such that $[\![\eta]\!]^\rho_\mu$ and b, c programs fulfilling the premises. The termination hypothesis implies that the non-truncated and truncated iterates have the same limit:

$$\Delta([\![(\mathbf{if}\ b\ \mathbf{then}\ c\ \mathbf{else\ skip})^n]\!]_\mu, [\![(\mathbf{if}\ b\ \mathbf{then}\ c\ \mathbf{else\ skip})^n; \mathbf{assert}\ \neg b]\!]_\mu) \xrightarrow{n\to\infty} 0$$

where Δ is the total variation distance defined by

$$\Delta(\mu, \mu') \triangleq \sum_m |\mu(m) - \mu'(m)|.$$

Since the limit of $[\![(\mathbf{if}\ b\ \mathbf{then}\ c\ \mathbf{else\ skip})^n; \mathbf{assert}\ \neg b]\!]_\mu$ is $[\![\mathbf{while}\ b\ \mathbf{do}\ s]\!]_\mu$, one can deduce that $[\![(\mathbf{if}\ b\ \mathbf{then}\ c\ \mathbf{else\ skip})^n]\!]_\mu$ is also a convergent sequence and has the same limit. We then conclude as in the proof of Proposition 5, using the t-closed property. □

We are now ready to prove soundness for the almost-sure termination rules. As noted before, the main challenge is proving termination; from there, we can conclude by rule $[While - G]$. Our arguments use basic notations and theorems from the theory of Markov processes; a full introduction to Markov theory is not feasible in this space. We collect some of the concepts we need here: - Positive recurrent; - Null recurrent; - Transient; - Stationary distribution; - Coupling; - Stochastic dominance; - Lifted Markov chain; - Absorbing state. These concepts all belong to the basic theory of Markov processes; the interested reader can consult a textbook [35] for a detailed introduction.

Proposition 7 (Soundness of rule [WHILE-PVB])**.** *Let η be a t-closed assertion. For any sub-distributions, valuations μ, ρ, such that $\mathfrak{C}_{\mathrm{PVB}}$ is valid. Then*

$$[\![\mathbf{while}\ b\ \mathbf{do}\ s]\!]\mu; \rho \models \eta.$$

Proof. By Proposition 6 and the premises, it suffices to prove almost-sure termination.

We consider the integer-valued variant $\tilde{e}$ as a random variable over the space of states. To this variant, we consider the family $(V_i)_i$ of random variables, where V_i represents the value taken by $\tilde{e}$ after the i-th iteration of the loop. The premises and the conditions

$$\{\mathcal{L} \wedge \Box(\tilde{e} = k \wedge 0 < k \leq K)\}\ s\ \{\mathcal{L} \wedge \Box(0 \leq \tilde{e} \leq K) \wedge \Pr[\tilde{e} < k] \geq \epsilon\}$$

ensure the following facts on the sequence V_i:

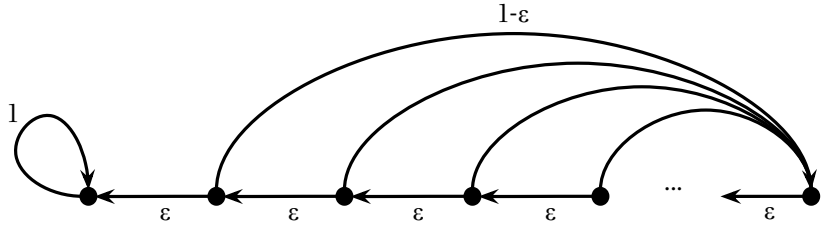
- The sequence is uniformly bounded by K.
- The probability of decreasing bounded below by ϵ:

$$\forall i \in \mathbb{N}.\ \Pr[V_i > V_{i+1} \mid V_i, \ldots, V_0] = \epsilon_i \geq \epsilon > 0.$$

The sketch of the proof is the following: we first introduce a Markov chain $(X_i)_i$ that reaches a particular state (the state zero) with probability 1. Then, we couple this chain with $(V_i)_i$ to show that $(U_i)_i$ stochastically dominates $(V_i)_i$. In particular, this shows that the probability of V_i (i.e., the variant) eventually reaching 0 is 1. The condition $\mathfrak{C}_{\mathrm{PVB}}$ guarantees that the loop guard is false when the variant hits 0, so almost-sure termination follows.

First, we can assume that $K > 0$ since if $K = 0$, the loop terminates immediately. Consider the following finite Markov chain $(X_i)_i$, over the state space $S = \{0, \ldots, K\}$ with probability of transitioning from state $a \in S$ to $b \in S$ given by

$$P_{a,b} \triangleq \begin{cases} \epsilon & \text{if } b = a - 1 \\ 1 - \epsilon & \text{if } b = K \\ 1 & \text{if } a = b = 0 \\ 0 & \text{otherwise.} \end{cases}$$

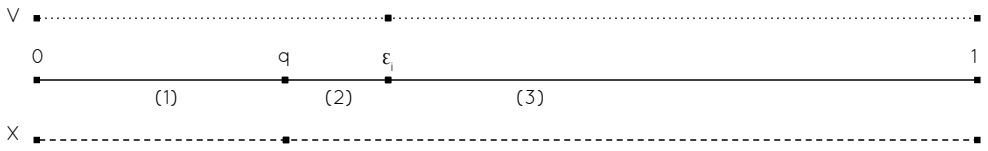


This chain models the following behavior: with probability ϵ the value decrease by one, while with probability $1-\epsilon$ it jumps to K. Since zero is the only terminating state, and that states from 1 to K is the only other strongly connected component of the underlying graph, the probability of terminating in the state zero is 1 by a standard result on Markov chains.

Let us now consider the coupling $(\tilde{X}_i, \tilde{V}_i)_i$ of the previous chain with the random walk $(V_i)_i$ given by randomness sharing. We take an i.i.d. family of uniform variables $(U_i)_i$ over $[0,1]$ and we consider the following behavior:

- Initially, $X_0 = V_0$.
- If $\tilde{X}_i = 0$ then $\tilde{X}_{i+1} = 0$ and if $\tilde{V}_i = 0$ then $\tilde{V}_{i+1} = 0$.
- Otherwise,
  1. If $U_i \leq \epsilon$ then $\tilde{X}_i = \tilde{X}_{i-1} - 1$ and $\tilde{V}_i$ is sampled like V_i, conditioned by the values already drawn — i.e. the events of $V_j = \tilde{V}_j$ for $j = 1, \dots, i-1$, and conditioning on the events where $\tilde{V}_i$ is lesser than the value taken by V_{i-1}.
  2. If $p < U_i \leq \epsilon_i$ then $\tilde{X}_i = K$ and $\tilde{V}_i$ is sampled as previously.
  3. If $\epsilon_i < U_i \leq 1$ then $\tilde{X}_i = K$ and $\tilde{V}_i$ is sampled as previously, but in this case, conditioned to the increasing events.

The following figure depicts the three cases depending on U_i.



It is easy to check that the first and second marginals of $(X_i, V_i)_i$ are given by $(X_i)_i$ and $(V_i)_i$ respectively, so we have constructed a coupling. Since we impose that $X_0 = V_0$ (initial position), a simple induction over i ensure that we have: $\Pr[\tilde{X}_i \leq \tilde{V}_i] = 1$ for all i. Therefore V_i is stochastically dominated by X_i, and thus we obtain that $\Pr[V_i > 0] \leq \Pr[X_i > 0]$, so $\lim_{i\to\infty} \Pr[V_i = 0] \geq 1 - \lim_{i\to\infty} \Pr[X_i > 0] = 1$ as desired. □

Proposition 8 (Soundness of the rule [WHILE-PVU])**.**

Proof. By Proposition 6 and the premises, it suffices to prove almost-sure termination.

We consider the integer-valued variant $\tilde{e}$ as a random variable over the space of states.[5] We model the evolution of this variant with a family $(V_i)_i$ of random variables, where V_i represents the value taken by $\tilde{e}$ after the i-th iteration of the loop. The premises and the conditions

The proof proceeds exactly like before: we introduce the sequence of $(V_i)_i$ to keep track of the values taken by the variant, then we introduce a Markov chain to reason stochastically and bound the behavior of the V_i and ultimately conclude by a coupling argument.

Like in the previous case, we introduce couple the sequence $(V_i)_i$ to a Markov chain in order to show almost-sure termination. We have by the premises that the probability of decreasing for the variant is greater than the probability of increasing. We can then lower bound the probability of strictly decreasing by a certain q and upper bound the probability increasing by p such that $q > p > 0$ (it is always possible thanks to side condition on the variant).

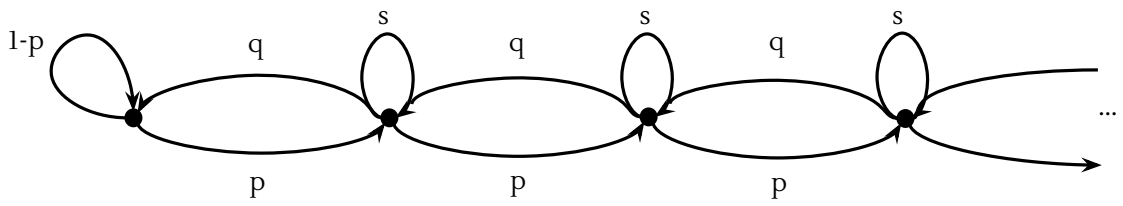
In this case, we define the Markov chain $(X_i)_i$ with state space $S = \mathbb{N}$, with probability of transitioning from $a \in S$ to $b \in S$ given by

$$\tilde{P}_{a,b} \triangleq \begin{cases} 1 & \text{if } a = b = 0 \\ q & \text{if } b = a - 1 \\ p & \text{if } b = a + 1 \text{ and } a > 0 \\ s & \text{if } a = b \neq 0 \\ 0 & \text{otherwise.} \end{cases}$$

This chain realizes a markovian approximation of the behavior of the variant. To facilitate the analysis of the previous chain we introduce the lifted chain P:

$$P_{a,b} \triangleq \begin{cases} 1-p & \text{if } a = b = 0 \\ q & \text{if } b = a - 1 \\ p & \text{if } b = a + 1 \\ s & \text{if } a = b \neq 0 \\ 0 & \text{otherwise.} \end{cases}$$

[5] This is possible thanks to the lossless requirements: the total weight remains invariant through the computation. We treat the particular case where the initial weight is 1, the general proof is done exactly in the same way by scaling.



The chain models the following behavior: at a given state, with probability p one increase by one, decrease by one with probability q and stay put with probability $s = 1 - p - q$. At the state 0, we have probability p of increasing by 1, and probability $1 - p$ of staying put.

This chain is irreducible. The analysis of chain $(X_i)_i$ is a bit more complicated than in the bounded case. We start by studying the existence of a *stationery distribution* π, that is, a distribution on S that satisfies

$$p \cdot \pi_{i-1} + q \cdot \pi_{i+1} + s \cdot \pi_i = \pi_i$$

for $i > 0$, and

$$p \cdot \pi_1 + (1 - p) \cdot \pi_0 \pi_0.$$

By rearranging, we need

$$p \cdot \pi_{i-1} + q \cdot \pi_{i+1} = (p + q)\pi_i \text{ for all } i > 0$$
$$q \cdot \pi_1 = p \cdot \pi_0.$$

The general solution is given by

$$\pi_i = \begin{cases} a + b(\frac{p}{q})^i, & \text{if } p \neq q \\ a' + b'i, & \text{otherwise.} \end{cases}$$

Therefore, since $\sum_i \pi_i$ is convergent, the only admissible solutions have the form:

$$\pi_i = \begin{cases} b(\frac{p}{q})^i & \text{if } p \neq q \\ a' & \text{otherwise.} \end{cases}$$

But since the π is a probability distribution over the states, the condition $\sum_{i=0}^{\infty} \pi_i = 1$ must hold. Trivially, the cases where $p \geq q$ are excluded. If $p < q$, we have:

$$1 = \sum_{i=0}^{\infty} b \left(\frac{p}{q}\right)^i = \frac{bq}{q - p}$$

thus $b = \frac{q-p}{p}$ and the stationary distribution is given by: $\pi_i = \frac{q-p}{p}(p/q)^i$.

By a standard result from the analysis of Markov chains, a chain with a stationary distribution is positive recurrent, so $q > p$ leads to a positive recurrent chain.

Let us study what happens otherwise. The general theory of countable states Markov chains gives the following condition:

Lemma 5. *An irreducible chain X_n of state space S is transient if and only if there exists a state i and a unique solution: $\alpha : S \to [0, 1]$ such that:*

$$\alpha_i = 1, \qquad \textit{and} \qquad \inf_S \{\alpha_j\} = 0, \qquad \textit{and} \qquad \alpha_j = \sum_{k \in S} P_{j,k}\alpha_k \quad \textit{for all } j \neq i.$$

With this lemma, we consider when the chain $(X_i)_i$. If the function α exists, it must satisfy

$$\alpha_j = q \cdot \alpha_{j-1} + p \cdot \alpha_{j+1} + s \cdot \alpha_j$$

for all $j > 0$. Reasoning like previously we obtain the solutions:

$$\alpha_i = \begin{cases} (1 - b) + b(\frac{q}{p})^i & \text{if } p \neq q \\ 1 + bi & \text{otherwise.} \end{cases}$$

The condition implies that taking $b = 0$ is not admissible. Furthermore, if $p = q$ no bounded possible solutions exists (since we have $b \neq 0$) and thus there is no solution. We thus assume from now on that $p \neq q$, that is to say: $p > q$. Taking $b = 1$ leads gives $\alpha_j = (q/p)^i$ which is a valid solution. Therefore we obtain the following classification:

- If $p > q$ the chain is transient.
- If $p = q$ the chain is null recurrent.
- If $p < q$ the chain is positive recurrent.

Thus we deduce that since $p \leq q$ the probability to reach the state zero is equal to 1. Trivially the result extends the chain $\tilde{P}$: two chains have the same behavior, except when reaching zero, when $\tilde{P}$ stops. We then conclude with a similar argument by coupling the variant $(V_i)_i$ to $\tilde{P}$. Thus, the variant reaches zero almost-surely, since in our particular case $p \leq q$. By the condition $\mathfrak{C}_{\mathrm{PVU}}$, the loop terminates when the variant reaches 0, so the loop terminates almost surely. This proves the termination of the loop thus the soundness of the rule. □

$$
\begin{array}{lcl}
\mathrm{pm}(\mathbf{skip}, p^\bullet) & := & p^\bullet \\
\mathrm{pm}(s_1; s_2, p^\bullet) & := & \mathrm{pm}(s_1, \mathrm{pm}(s_2, p^\bullet)) \\
\mathrm{pm}(x := e, p^\bullet) & := & p^\bullet[e/x] \\
\mathrm{pm}(x \xleftarrow{\$} \mathcal{D}(e), p^\bullet) & := & \mathcal{P}_x^{\mathcal{D}(e)}(p^\bullet) \\
\mathrm{pm}(\mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2, p^\bullet) & := & \mathrm{pm}(s_1, p^\bullet)_{|e} + \mathrm{pm}(s_2, p^\bullet)_{|\neg e} \\
\mathrm{pm}(\mathbf{abort}, p^\bullet) & := & 0 \\
\mathrm{pm}(s, o(\overline{p})) & := & o(\overline{\mathrm{pm}(s, p)}) \\
\mathrm{pm}(s, c) & := & c \\
 & & \\
\mathrm{pc}(s, q(\overline{p})) & := & q(\overline{\mathrm{pm}(s, p)}) \\
\mathrm{pc}(s, \eta_1\ S\ \eta_2) & := & \mathrm{pc}(s, \eta_1)\ S\ \mathrm{pc}(s, \eta_2) \\
\mathrm{pc}(s, \forall_{\dot{y}\in\mathbb{Z}}.\ \eta) & := & \forall_{\dot{y}\in\mathbb{Z}}.\ \mathrm{pc}(s, \eta) \\
\mathrm{pc}(s, \exists_{\dot{y}\in\mathbb{Z}}.\ \eta) & := & \exists_{\dot{y}\in\mathbb{Z}}.\ \mathrm{pc}(s, \eta)
\end{array}
$$

For $p^\bullet$ an integral, c constant, $o \in \mathbf{Ops}, q \in \mathbf{Pred}, S \in \{\wedge, \vee, \Rightarrow\}$.

Figure 6. Calculus of the preconditions

B. Precondition calculus

In all program logics, a procedure to compute *preconditions* is a powerful tool that can make the logic easier to use. Given an assertion η and a statement s, we wish to find an assertion η^* that is the precondition of s that implies η as a postcondition. Our plan for computing preconditions will be to look at each atomic expression p inside η—the integral expressions. We will replace each p by an expression p^* that has the same denotation before running the s as p after running s. This yields an assertion η^* that, interpreted before running s, is logically equivalent to η interpreted after running s.

Let the *preterm* of a probability expression η with regards to a program s be a probability expression $\mathrm{pm}(s, \eta)$ that, evaluated at an initial state, has the same denotation as η after executing s on the initial state. Formally, we require

$$[\![\mathrm{pm}(s, \eta)]\!]^\rho_\mu = [\![\eta]\!]^\rho_{[\![s]\!]_\mu}$$

for all probabilistic states μ and substitutions ρ.

Once we have defined the preterms, we can lift them to define the precondition $\mathrm{pc}\, s, \eta$ of a probability assertion η with regards to a program s, by applying the operator pm to each generalized expression appearing in η. The preterms and preconditions are constructed by induction, according to Figure 6. (Note that the precondition is not defined for looping commands, similar to the situation for deterministic languages.)

While preconditions in typical Hoare logics are defined by induction on the *statement*, our preconditions are defined by induction on the *formula*. The essential reason for this choice is the rule for conditional statements. Recall that the rule [IF] does not allow postconditions of arbitrary shape: the postcondition must have the form $\eta_1 \oplus \eta_2$. If we are to find the precondition of a general assertion η, we first need to guess formulas η_1, η_2 such that $\eta_1 \oplus \eta_2 \implies \eta$. This problem is also present if we want to apply rule [IF] directly.

In contrast, by defining precondition in terms of preterms, we avoid this problem since we do not need to split the assertion to compute the preterm. Formally, the definition for the preterm of a conditional statement is:

$$\mathrm{pm}(\mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2, p^\bullet) \triangleq \mathrm{pm}(s_1, p^\bullet)_{|e} + \mathrm{pm}(s_2, p^\bullet)_{|\neg e},$$

where $p^\bullet = \oint_\Gamma \tilde{e}$ is an atomic probability expression, and the conditioning operation $p_{|e}$ is defined by substitution, with base cases

$$\oint_\Gamma \tilde{e}\ _{|e} \triangleq \oint_\Gamma \mathbf{1}_{e=\top} \times \tilde{e} \quad \text{and} \quad c_{|e} \triangleq c.$$

To understand the preterm definition, note that $\mathrm{pm}(s_1, p^\bullet)$ and $\mathrm{pm}(s_2, p^\bullet)$ represent the probability expression $p^\bullet$ before executing s_1 and s_2 in isolation, respectively. Since the final expression $p^\bullet$ includes contributions from both branches, we scale each preterm by the probability that we enter the respective branch—this is the effect of the conditioning operator.

C. Derived rules

In this section, we prove soundness for the derived rules.

C.1 Deterministic rules

We will use a derived rule for the null sub-distribution:

$$\textsc{Null}\ \frac{}{\{\Box\bot\}\ s\ \{\Box\bot\}}.$$

Proposition 9 (Soundness of rule [NULL])**.** *Rule [*NULL*] is a derived rule, and therefore sound.*

Proof. Induction on the structure of s. □

Now, we can prove that the deterministic branching rule is sound.

Proposition 10 (Soundness of rule [IF-D])**.** *The rule [*IF-D*] is a derived rule, and therefore sound.*

Proof. Recall the rule:

$$\textsc{If-D}\ \frac{\{\eta \wedge \Box e\}\ s_1\ \{\eta'\} \qquad \{\eta \wedge \Box \neg e\}\ s_2\ \{\eta'\}}{\{\eta \wedge \circledcirc e\}\ \mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2\ \{\eta'\}}$$

Let μ be a probabilistic state, and let ρ be a logical valuation satisfying the precondition of the conclusion. Now, either $\mu; \rho \models \Box e$ or $\mu; \rho \models \Box \neg e$. Without loss of generality, suppose we are in the first case.

We apply the general rule [IF]:

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

with preconditions $\eta_1 \triangleq \eta \wedge \Box e$ and $\eta_2 \triangleq \Box \bot$, and post-conditions $\eta_1' \triangleq \eta'$ and $\eta_2' \triangleq \Box \bot$.

By assumption we have the first premise. By rule [NULL] we have the second premise. Thus, we have

$$[\![\mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2]\!]_\mu; \rho \models \eta' \oplus \Box \bot.$$

But $\eta' \oplus \Box \bot$ implies η', and so we are done.

The case with $\mu; \rho \models \Box \neg e$ proceeds essentially the same. □

We can also prove soundness of the deterministic looping rule.

Proposition 11 (Soundness of rule [WHILE-D])**.** *Rule [*WHILE-D*] is sound.*

Proof. Recall the rule:

$$\textsc{While-D}\ \frac{\{\eta \wedge \Box b\}\ s\ \{\eta \wedge \circledcirc b\}}{\{\eta \wedge \circledcirc b\}\ \mathbf{while}\ b\ \mathbf{do}\ s\ \{\eta \wedge \Box \neg b\}}$$

Note that [WHILE-D] is a trivially derivable from [WHILE-C] if there is a program expression that can serve as a decreasing variant. If not, the proof proceeds as in Proposition 4. □

C.2 Independence rules

Lemma 6 (Soundness of rule [SAMPLE-IND])**.** *For any state μ, interpretation ρ, disjoint variables $x, x_1, \ldots, x_n$ such that $\mu; \rho \models \#\langle x_1, \ldots, x_n\rangle$ then*

$$[\![x \xleftarrow{\$} g]\!]_\mu; \rho \models \#\langle x, x_1, \ldots, x_n\rangle$$

where g is a primitive distribution (has no variables).

Proof. The case where μ has weight 0 (the null sub-distribution) is trivial, so we will assume that $[\![\oint \top]\!]^\rho_\mu > 0$ in the remainder. Let t be a particular memory, and let μ' be the sub-distribution $[\![x \xleftarrow{\$} \mathcal{D}]\!]^\rho_m$. We have:

$$\begin{aligned}\mu'(t) &= \sum_m \sum_v \mathbf{1}_{m[x:=v]=t} \mu(m) [\![\mathcal{D}]\!]_m(v) \\ &= [\![g]\!](t(x)) \sum_{m:m=_x t} \mu(m)\end{aligned}$$

where $=_x$ refers to the equality on all memory cells except on x. Note that the denotation of g is independent of the memory, since it is a closed expression. Thus when computing the measure of the set of memories

$$E \triangleq \{m : (m(x) = \alpha \wedge m(x_1) = \alpha_1 \wedge \ldots \wedge m(x_n) = \alpha_n)\}$$

we have in μ':

$$\begin{aligned}\mu'(E) &= [\![g]\!](\alpha) \sum_m \mathbf{1}_{m(x_1)=\alpha_1,\ldots,m(x_n)=\alpha_n} \mu(m) \\ &= [\![g]\!](\alpha) \mu(E') \\ &= \frac{[\![\Pr[x = a]]\!]^\rho_{\mu'}}{[\![\Pr[\top]]\!]^\rho_\mu} \cdot \mu(E') \\ &= \frac{[\![\Pr[x = a]]\!]^\rho_{\mu'}}{[\![\Pr[\top]]\!]^\rho_{\mu'}} \cdot \mu(E')\end{aligned}$$

where

$$E' \triangleq \{m : (m(x_1) = \alpha_1 \wedge \ldots \wedge m(x_n) = \alpha_n)\}$$

and the last step used the fact that the sampling operation is lossless, i.e., the denotation of g is a proper sub-distribution. Since x is disjoint from $x_1, \ldots, x_n$, we can conclude by the independence of the x_i (again using losslessness of sampling). □

This proof generalized naturally in the case where the sampling distributions are functions of a set of variables $y_1, \ldots y_n$ independent of the $x_1, \ldots, x_n$.