A program logic for union bounds

Gilles Barthe[1], Marco Gaboardi[2], Benjamin Grégoire[3], Justin Hsu[4], and Pierre-Yves Strub[1]

1 **IMDEA Software Institute**
2 **University at Buffalo, SUNY**
3 **Inria Sophia Antipolis - Méditerranée**
4 **University of Pennsylvania**

Abstract

We propose a probabilistic Hoare logic aHL based on the union bound, a tool from basic probability theory. While the union bound is simple, it is an extremely common tool for analyzing randomized algorithms. In formal verification terms, the union bound allows flexible and compositional reasoning over possible ways an algorithm may go wrong. It also enables a clean separation between reasoning about probabilities and reasoning about events, which are expressed as standard first-order formulas in our logic. Notably, assertions in our logic are non-probabilistic, even though we can conclude probabilistic facts from the judgments.

Our logic can also prove accuracy properties for interactive programs, where the program must produce intermediate outputs as soon as pieces of the input arrive, rather than accessing the entire input at once. This setting also enables adaptivity, where later inputs may depend on earlier intermediate outputs. We show how to prove accuracy for several examples from the differential privacy literature, both interactive and non-interactive.

1998 ACM Subject Classification D.2.4 Software/Program Verification

Keywords and phrases Probabilistic Algorithms, Accuracy, Formal Verification, Hoare Logic, Union Bound

Digital Object Identifier 10.4230/LIPIcs.ICALP.2016.♯

1 Introduction

Probabilistic computations arise naturally in many areas of computer science. For instance, they are widely used in cryptography, privacy, and security for achieving goals that lie beyond the reach of deterministic programs. However, the correctness of probabilistic programs can be quite subtle, often relying on complex reasoning about probabilistic events.

Accordingly, probabilistic computations present an attractive target for formal verification. A long line of research, spanning more than four decades, has focused on expressive formalisms for reasoning about general probabilistic properties both for purely probabilistic programs and for programs that combine probabilistic and non-deterministic choice (see, e.g., [29, 34, 35]).

More recent research investigates specialized formalisms that work with more restricted assertions and proof techniques, aiming to simplify formal verification. As perhaps the purest examples of this approach, some program logics prove probabilistic properties by working purely with non-probabilistic assertions; we call such systems *lightweight* logics. Examples include *probabilistic relational Hoare logic* [3] for proving the reductionist security of cryptographic constructions, and the related *approximate probabilistic relational Hoare logic* [4] for reasoning about differential privacy. These logics rely on the powerful abstraction of *probabilistic couplings* to derive probabilistic facts from non-probabilistic assertions [7].

43rd International Colloquium on Automata, Languages, and Programming (ICALP 2016).
Editors: Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi; Article No. ♯; pp. ♯:1–♯:20
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Lightweight logics are appealing because they can leverage ideas for verifying deterministic programs, a rich and well-studied area of formal verification. However, existing lightweight logics apply only to relational verification: properties about the relation between two programs. In this paper, we propose a non-relational, lightweight logic based on the *union bound*, a simple tool from probability theory. For arbitrary properties $E_1, \ldots, E_n$, the union bound states that

$$\Pr\left[\cup_{i=1}^{n} E_i\right] \leq \sum_{i=1}^{n} \Pr[E_i] \ .$$

Typically, we think of the events E_i as *bad events*, describing different ways that the program may fail to satisfy some target property. Bad events can be viewed as propositions on single program states, so they can be represented as non-probabilistic assertions. For example, the formula $x > 10$ defines a bad event for x a program variable. If x stores the result from a random sample, this bad event models when the sample is bigger than 10. The union bound states that no bad events happen, except with probability at most the sum of the probabilities of each bad event.

The union bound is a ubiquitous tool in pen-and-paper proofs due to its flexible and compositional nature: to bound the probability of a collection of failures, consider each failure in isolation. This compositional style is also a natural fit for formal verification. To demonstrate this, we formalize a Hoare logic aHL based on the union bound for a probabilistic imperative language. The assertions in our logic are non-probabilistic, but judgments carry a numeric index for tracking the failure probability. Concretely, the aHL judgment

$$\vdash_{\beta} c : \Phi \Longrightarrow \Psi$$

states that every execution of a program c starting from an initial state satisfying Φ yields a distribution in which Ψ holds except with probability at most β. We define a proof system for the logic and show its soundness. We also define a sound embedding of aHL into standard Hoare logic, by instrumenting the program with ghost code that tracks the index β in a special program variable. This is a useful reduction that also applies to other lightweight logics [5].

Moreover, our logic applies both to standard algorithms and to *interactive* algorithms, a richer class of algorithms that is commonly studied in contexts such as *online learning* (algorithms which make predictions about the future input) and *streaming* (algorithms which operate on datasets that are too large to fit into memory by processing the input in linear passes). Informally, interactive algorithms receive their input in a sequence of chunks, and must produce intermediate outputs as soon as each chunk arrives. In some cases the input can be *adaptive*: later inputs may depend on earlier outputs. Besides enabling new classes of algorithms, interactivity allows more modularity. We can decompose programs into interacting parts, analyze each part in isolation, and reuse the components.

We demonstrate aHL on several algorithms satisfying *differential privacy* [15], a statistical notion of privacy which trades off between the privacy of inputs and the accuracy of outputs. Prior work on verifying private algorithms focuses on the privacy property for non-interactive algorithms (see, e.g. [4, 18, 37]). We provide the first verification of *accuracy* for both non-interactive and interactive algorithms. We note however that aHL, like the union bound, can be applied to a wide range of probabilistic programs beyond differential privacy.

2 A union bound logic

Before introducing the program logic, we will begin by reviewing a largely standard, probabilistic imperative language. We state the soundness of the logic and describe the embedding into Hoare logic. The semantics of the language and the proof of soundness are deferred to the appendix.

2.1 Language

We will work with a core imperative language with a command for random sampling from distributions, and procedure calls. The set of commands is defined as follows:

$$
\begin{array}{rlll}
\mathcal{C} ::= & \mathsf{skip} & & \text{noop} \\
\mid & \mathcal{X} \leftarrow \mathcal{E} & & \text{deterministic assignment} \\
\mid & \mathcal{X} \xleftarrow{\$} \mathcal{D}(\mathcal{E}) & & \text{probabilistic assignment} \\
\mid & \mathcal{C}; \mathcal{C} & & \text{sequencing} \\
\mid & \mathsf{if}\ \mathcal{E}\ \mathsf{then}\ \mathcal{C}\ \mathsf{else}\ \mathcal{C} & & \text{conditional} \\
\mid & \mathsf{while}\ \mathcal{E}\ \mathsf{do}\ \mathcal{C} & & \text{while loop} \\
\mid & \mathcal{X} \leftarrow \mathcal{F}(\mathcal{E}) & & \text{procedure call} \\
\mid & \mathcal{X} \leftarrow \mathcal{A}(\mathcal{E}) & & \text{external call}
\end{array}
$$

Here, $\mathcal{X}$ is a set of *variables*, $\mathcal{E}$ is a set of *expressions*, and $\mathcal{D}$ is a set of *distribution constructors*, which can be parameterized by standard expressions. Variables and expressions are typed, ranging over booleans, integers, lists, etc. The expression grammar is entirely standard, and we omit it.

We distinguish two kinds of procedure calls: $\mathcal{A}$ is a set of external procedure names, and $\mathcal{F}$ is a set of internal procedure names. We assume we have access to the code of internal procedures, but not the code of external procedures. We think of external procedures as controlled by some external *adversary*, who can select the next input in an interactive algorithm. Accordingly, external procedures run in an *external memory* separate from the main program memory, which is shared by all internal procedures.

For simplicity, procedures take a single argument, do not have local variables, and are not mutually recursive. A program consists of a sequence of procedures definitions, each of the following form:

$$\mathbf{proc}\ f(\mathbf{arg}_f)\{c; \mathbf{return}\ r;\}$$

Here, f is a procedure name, $\mathbf{arg}_f \in \mathbf{Vars}$ is the formal argument of f, c is the function body and r is its return value. We assume that distinct procedure definitions do not bind the same procedure name and that the program variable $\mathbf{arg}_f$ can only appear in the body of f.

Before we define the program semantics, we first need to introduce a few definitions from probability theory.

▶ **Definition 1.** A discrete sub-distribution over a set A is defined by a *mass function* $\mu : A \to [0, 1]$ such that:

- the support $\mathrm{supp}(\mu)$ of μ—defined as $\{x \in A \mid \mu(x) \neq 0\}$—is countable; and
- the weight $\mathrm{wt}(\mu)$ of μ—defined as $\sum_{x\in A} \mu(x)$—satisfies $\mathrm{wt}(\mu) \leq 1$.

A *distribution* is a sub-distribution with weight 1. The probability of an event P w.r.t. μ, written $\Pr_\mu[P]$ (or $\Pr[P]$ when μ is clear from the context), is defined as $\sum_{x\in A\mid P(x)} \mu(x)$. When Φ is an assertion (assuming that $A \equiv \mathsf{State}$), we write $\Pr_\mu[\Phi]$ for $\Pr_\mu[\lambda m.\, m \models \Phi]$. Likewise, when $v \in A$, we write $\Pr_\mu[v]$ for $\Pr_\mu[\lambda x.\, x = v]$.

Commands are interpreted as a function from memories to sub-distributions over memories, where memories are finite maps from program and external variables to values. More formally, if State is the set of memories then the interpretation of c, written $[\![c]\!]$, is a function from

State to $\mathbf{Distr}(\mathsf{State})$, where $\mathbf{Distr}(\mathsf{T})$ denotes the set of discrete sub-distributions over T. The definition of $[\![c]\!]$ enforces the separation between the internal and external states—only commands performing external procedure calls can act on the external memory. The interpretation of external procedure calls is parameterized by functions—one for each external procedure—of type $\mathsf{State}_{|\mathcal{A}} \to \mathbf{Distr}(\mathsf{State}_{|\mathcal{A}})$, where $\mathsf{State}_{|\mathcal{A}}$ is the set of memories *restricted* to the external variables. Thus, external procedures can only access the external memory.

2.2 Logic

Now that we have seen the programs, let us turn to the program logic. Our judgments are similar to standard Hoare logic with an additional numeric index representing the probability of failure. Concretely, the judgments are of the following form:

$$\vdash_\beta c : \Phi \Longrightarrow \Psi$$

where Φ and Ψ are first-order formulas over the program variables representing the pre- and post-condition, respectively. We stress that Φ and Ψ are *non-probabilistic* assertions: they do not mention the probabilities of specific events, and will be interpreted as properties of individual memories rather than distributions over memories. This is reflected by the validity relation for assertions: $m \models \Phi$ states that Φ is valid in the *single* memory m, rather than in a distribution over memories. Similarly, $\models \Phi$ states that Φ is valid in all (single) memories. By separating the assertions from the probabilistic features of our language, the assertions are simpler and easier to manipulate. The index β is a non-negative real number (typically, from the unit interval $[0, 1]$).

Now, we can define semantic validity for our judgments. In short, the index β will be an upper bound on the probability that the postcondition Ψ does not hold on the output distribution, assuming the precondition Φ holds on the initial memory.

▶ **Definition 2** (Validity). A judgment $\vdash_\beta c : \Phi \Longrightarrow \Psi$ is *valid* if for every memory m such that $m \models \Phi$, we have:

$$\Pr_{[\![c]\!](m)}[\neg\Psi] \leq \beta.$$

We present the main proof rules of our logic in Figure 1. The rule for random sampling [Rand] allows us to assume a proposition Ψ about the random sample provided that Ψ fails with probability at most β. This is a semantic condition which we introduce as an axiom for each primitive distribution.

The remaining rules are similar to the standard Hoare logic rules, with special handling for the index. The sequence rule [Seq] states that the failure probabilities of the two commands add together; this is simply the union bound internalized in our logic. The conditional rule [If] assumes that the indices for the two branch judgments are equal—which can always be achieved via weakening—keeping the same index for the conditional. Roughly, this is because only one branch of the conditional is executed. The loop rule [While] simply accumulates the failure probability β throughout the iterations; the side conditions ensure that the loop terminates in at most k iterations except with probability $k \cdot \beta$. To reason about procedure calls, standard (internal) procedure calls use the rule [Call], which substitutes the argument and return variables in the pre- and post-condition, respectively. External procedure calls use the rule [Ext]. We do not have access to the implementation of the procedure; we know just the type of the return value.

The structural rules are also similar to the typical Hoare logic rules. The weakening rule [Weak] allows strengthening the precondition and weakening the postcondition as usual, but also allows increasing the index—this corresponds to allowing a possibly higher probability of

$$\frac{}{\vdash_0 \mathsf{skip} : \Phi \Longrightarrow \Phi}\ [\textsc{Skip}] \qquad \frac{}{\vdash_0 x \leftarrow e : \Phi[e/x] \Longrightarrow \Phi}\ [\textsc{Assn}]$$

$$\frac{\forall m.\, m \models \Phi \implies \Pr_{[\![x \xleftarrow{\$} d(e)]\!](m)}[\neg\Psi] \leq \beta}{\vdash_\beta x \xleftarrow{\$} d(e) : \Phi \Longrightarrow \Psi}\ [\textsc{Rand}]$$

$$\frac{\begin{array}{c}\vdash_\beta c : \Phi \Longrightarrow \Phi' \\ \vdash_{\beta'} c' : \Phi' \Longrightarrow \Phi''\end{array}}{\vdash_{\beta+\beta'} c; c' : \Phi \Longrightarrow \Phi''}\ [\textsc{Seq}] \qquad \frac{\begin{array}{c}\vdash_\beta c : \Phi \wedge e \Longrightarrow \Psi \\ \vdash_\beta c' : \Phi \wedge \neg e \Longrightarrow \Psi\end{array}}{\vdash_\beta \mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c' : \Phi \Longrightarrow \Psi}\ [\textsc{If}]$$

$$\frac{\begin{array}{c} e_v : \mathbb{N} \qquad \models \Phi \wedge e_v \leq 0 \to \neg e \\ \vdash_\beta c : \Phi \Longrightarrow \Phi \qquad \forall \eta > 0.\ \vdash_0 c : \Phi \wedge e \wedge e_v = \eta \Longrightarrow e_v < \eta \end{array}}{\vdash_{k \cdot \beta} \mathsf{while}\ e\ \mathsf{do}\ c : \Phi \wedge e_v \leq k \Longrightarrow \Phi \wedge \neg e}\ [\textsc{While}]$$

$$\frac{\begin{array}{c}\mathbf{proc}\ f(\mathbf{arg}_f)\{c; \mathbf{return}\ r;\} \\ \vdash_\beta c : \Phi \Longrightarrow \Psi[r/\mathbf{res}_f]\end{array}}{\vdash_\beta x \leftarrow f(e) : \Phi[e/\mathbf{arg}_f] \Longrightarrow \Psi[x/\mathbf{res}_f]}\ [\textsc{Call}] \qquad \frac{}{\vdash_0 x \leftarrow f(e) : \forall v.\ \Psi[v/x] \Longrightarrow \Psi}\ [\textsc{Ext}]$$

$$\frac{\begin{array}{c}\models \Phi' \to \Phi \qquad \models \Psi \to \Psi' \qquad \beta \leq \beta' \\ \vdash_\beta c : \Phi \Longrightarrow \Psi\end{array}}{\vdash_{\beta'} c : \Phi' \Longrightarrow \Psi'}\ [\textsc{Weak}] \qquad \frac{c \text{ does not modify variables in } \Phi}{\vdash_0 c : \Phi \Longrightarrow \Phi}\ [\textsc{Frame}]$$

$$\frac{\begin{array}{c}\vdash_\beta c : \Phi \Longrightarrow \Psi \\ \vdash_{\beta'} c : \Phi \Longrightarrow \Psi'\end{array}}{\vdash_{\beta+\beta'} c : \Phi \Longrightarrow \Psi \wedge \Psi'}\ [\textsc{And}] \qquad \frac{\begin{array}{c}\vdash_\beta c : \Phi \Longrightarrow \Psi \\ \vdash_\beta c : \Phi' \Longrightarrow \Psi\end{array}}{\vdash_\beta c : \Phi \vee \Phi' \Longrightarrow \Psi}\ [\textsc{Or}] \qquad \frac{}{\vdash_1 c : \Phi \Longrightarrow \bot}\ [\textsc{False}]$$

Figure 1 Selected proof rules.

failure. The frame rule [Frame] preserves assertions that do not mention variables modified by the command. The conjunction rule [And] is another instance of the union bound, allowing us to combine two postconditions while adding up the failure probabilities. The case rule [Or] is the dual of [And] and takes the maximum failure probability among two post-conditions when taking their disjunction. Finally, the rule [False] allows us to conclude false with failure probability 1: With probability at most 0, false holds in the final memory.

We can show that our proof system is sound with respect to the semantics; the proof is deferred to the appendix.

▶ **Theorem 3** (Soundness). *All derivable judgments $\vdash_\beta c : \Phi \Longrightarrow \Psi$ are valid.*

In addition, we can define a sound embedding into Hoare logic in the style of Barthe et al. [5]. Assuming a fresh program variable x_β of type $\mathbb{R}$, we can transform a command c such that $\vdash_\beta c : \Phi \Longrightarrow \Psi$ to a new command $\lceil c \rceil$ and a proof of the standard Hoare logic judgment

$$\vdash \lceil c \rceil : \Phi \wedge x_\beta = 0 \implies \Psi \wedge x_\beta \leq \beta\ .$$

The command $\lceil c \rceil$ is obtained from c by replacing all probabilistic sampling $x \xleftarrow{\$} d(e)$ with a call to an abstract, non-probabilistic procedure call $x \leftarrow \mathrm{Sample}^\diamond\,(d(e))$, whose specification models the postcondition of [Rand]:

$$\frac{\forall m.\, m \models \Phi \implies \Pr_{[\![x \xleftarrow{\$} d(e)]\!](m)} [\neg \Psi] \le \iota}{\vdash x \leftarrow \mathrm{Sample}^{\diamond}\left(d(e)\right) : \Phi \wedge x_{\beta} \le \nu \implies \Psi \wedge x_{\beta} \le \nu + \iota} \; .$$

3 Accuracy for differentially private programs

Now that we have presented our logic aHL, we will follow by verifying several examples. Though our system applies to programs from many domains, we will focus on programs satisfying *differential privacy*, a statistical notion of privacy proposed by Dwork et al. [15]. At a very high level, these programs take private data as input and add random noise to protect privacy. (Interested readers should consult a textbook [14] for a more detailed presentation.) In contrast to existing formal verification work, which verifies the privacy property, we will verify *accuracy*. This is just as important as privacy: the constant function is perfectly private but not very useful.

All of our example programs take samples from the Laplace distribution.

▶ **Definition 4.** The *(discrete) Laplace* distribution $\mathcal{L}_{\epsilon}(e)$ is parameterized by a scale parameter $\epsilon > 0$ and a mean e. The distribution ranges over the real numbers $\{\nu = k + e\}$ for k an integer, releasing ν with probability proportional to:

$$\Pr_{\mathcal{L}_{\epsilon}(e)}[\nu] \propto \exp\left(-\epsilon \cdot |\nu - e|\right).$$

This distribution satisfies a basic accuracy property.

▶ **Lemma 5.** *Let $\beta \in (0,1)$, and let ν be a sample from the distribution $\mathcal{L}_{\epsilon}(e)$. Then,*

$$\Pr_{\mathcal{L}_{\epsilon}(e)}\left[|\nu - e| > \tfrac{1}{\epsilon} \log \tfrac{1}{\beta}\right] < \beta \; .$$

Thus, the following sampling rule is sound for our system for every $\beta \in (0,1)$:

$$\frac{}{\vdash_{\beta} x \xleftarrow{\$} \mathcal{L}_{\epsilon}(e) : \top \implies |x - e| \le \dfrac{1}{\epsilon} \log \dfrac{1}{\beta}} \; [\textsc{LapAcc}]$$

Before presenting the examples, we will set some common notations and terminology. First, we consider a set db of databases,[1] a set query of queries, and primitive functions

$$\begin{array}{rcl}
\mathsf{evalQ} & : & \mathsf{query} \to \mathsf{db} \to \mathbb{R} \\
\mathsf{invQ} & : & \mathsf{query} \to \mathsf{query} \\
\mathsf{negQ} & : & \mathsf{query} \to \mathsf{query} \\
\mathsf{size} & : & \mathsf{db} \to \mathbb{N} \\
\mathsf{error} & : & \mathsf{query} \to \mathsf{db} \to \mathsf{query}
\end{array}$$

satisfying

$$\begin{array}{rcl}
\mathsf{evalQ}(\mathsf{invQ}(q), d) & = & -\mathsf{evalQ}(q, d) \\
\mathsf{evalQ}(\mathsf{negQ}(q), d) & = & \mathsf{size}(d) - \mathsf{evalQ}(q, d) \\
\mathsf{evalQ}(\mathsf{error}(q, d_1), d_2) & = & \mathsf{evalQ}(q, d_1) - \mathsf{evalQ}(q, d_2)
\end{array}$$

Concretely, one can identify query with the functions db → $\mathbb{R}$ and obtain an easy realization of the above functions and axioms.

In some situations, we may need additional structure on the queries to prove the accuracy guarantees. In particular, a query q is *linear* if

[1] The general setting of differential privacy is that the database contains private information that must be protected. However, this fact will not be important for proving accuracy.

- for every two databases d, d', we have $q(d + d') = q(d) + q(d')$ for a commutative and associative operator $+$ on databases; and
- for the database d_0 that is the identity of $+$, we have $q(d_0) = 0$.

Concretely, we can identify db with the set of multisets, $+$ with multiset union, and d_0 with the empty multiset.

3.1 Report-noisy-max

Our first example is the *Report-noisy-max* algorithm (see, e.g., Dwork and Roth [14]). Report-noisy-max is a variant of the *exponential mechanism* [32], which provides the standard way to achieve differential privacy for computations whose outputs lie in a finite (perhaps non-numeric) set $\mathcal{R}$. Both algorithms perform the same computations, except that the exponential mechanism adds *one-sided* Laplace noise whereas Report-noisy-max adds regular Laplace noise. Thus, accuracy for both algorithms is verified in essentially the same way. We focus on Report-noisy-max to avoid defining one-sided Laplace.

Report-noisy-max finds an element of a finite set $\mathcal{R}$ that approximately maximizes some *quality score* function qscore, which takes as input an element $r \in \mathcal{R}$ and a database d. Operationally, Report-noisy-max computes the quality score for each element of $\mathcal{R}$, adds Laplace noise, and returns the element with the highest (noisy) value. We can implement this algorithm with the following code, using syntactic sugar for arrays:

```
proc $\mathrm{RNM}(\mathcal{R}, d)$ :
  $flag \leftarrow 1; best \leftarrow 0;$
  while $\mathcal{R} \neq \emptyset$ do
    $r \leftarrow \mathsf{pick}(\mathcal{R}); noisy[r] \xleftarrow{\$} \mathcal{L}_{\epsilon/2}(\mathsf{qscore}(r, d));$
    if $(noisy[r] > best \lor flag = 1)$ then
      $flag \leftarrow 0; r^* \leftarrow r; best \leftarrow noisy[r];$
    $\mathcal{R} \leftarrow \mathcal{R} \setminus \{r\};$
return $r^*$;
```

The scale $\epsilon/2$ of the Laplace distribution ensures an appropriate level of differential privacy under certain assumptions; we will not discuss privacy in the remainder.

▶ **Theorem 6.** *Let $\beta \in (0, 1)$, and let $res \in \mathcal{R}$ be the output of Report-noisy-max on input d and quality score* qscore*. Then, we have the following judgment:*

$$\vdash_\beta \mathrm{RNM} : \top \Longrightarrow \forall r \in \mathcal{R}.\ \mathsf{qscore}(res, d) > \mathsf{qscore}(r, d) - \tfrac{4}{\epsilon} \log \tfrac{|\mathcal{R}|}{\beta}.$$

where $|\mathcal{R}|$ denotes the size of $\mathcal{R}$. This corresponds to the existing accuracy guarantee for Report-noisy-max (see, e.g., Dwork and Roth [14]).

Roughly, this theorem states that while the result res may not be the element with the absolute highest quality score, its quality score is not far below the quality score of any other element. For a brief sanity check, note that the guarantee weakens as we increase the range $\mathcal{R}$, or decrease the failure probability β.

The proof of accuracy is based on an instantiation of the rule [LapAcc] with e set to $\mathsf{qscore}(r, d)$, β set to $\beta/|\mathcal{R}|$, and ϵ set to $\epsilon/2$. First, we can show

$$\vdash_{\beta/|\mathcal{R}|} c : \top \Longrightarrow |noisy[r] - \mathsf{qscore}(r, d)| < \tfrac{2}{\epsilon} \log \tfrac{|\mathcal{R}|}{\beta}$$

where c is the loop body. Since the loop runs for $|\mathcal{R}|$ iterations, we also have

$$\vdash_\beta \mathrm{RNM} : \top \Longrightarrow \forall r \in \mathcal{R}.\ |noisy[r] - \mathsf{qscore}(r, d)| < \tfrac{2}{\epsilon} \log \tfrac{|\mathcal{R}|}{\beta}.$$

In order to prove this judgment, the loop invariant quantifies over all previously seen $r \in \mathcal{R}$. Combined with a straightforward invariant showing that r^* stores the index of the current maximum (noisy) score, the above judgment suffices to prove the accuracy guarantee for Report-noisy-max (Theorem 6).

3.2 Sparse Vector algorithm

Our second example is the *Sparse Vector algorithm*, which indicates which numeric queries take value (approximately) above some threshold value (see, e.g., Dwork and Roth [14]). Simpler approaches can accomplish this task by releasing the noisy answer to all queries and then comparing with the threshold, but the resulting error then grows linearly with the total number of queries. Sparse Vector does not release the noisy answers, but the resulting error grows only *logarithmically* with the total number of queries—a substantial improvement. The differential privacy property of Sparse Vector was recently formally verified [8]; here, we consider the accuracy property.

In the non-interactive setting, the algorithm takes as input a list of queries $q_1, q_2, \dots,$ a database d, and a numeric threshold $t \in \mathbb{R}$.[2] First, we add Laplace noise to the threshold t to calculate the noisy threshold T. Then, we evaluate each query q_i on d, add Laplace noise, and check if the noisy value exceeds T. If so, we output $\top$; if not, we output $\bot$.

Sparse Vector also works in the *interactive* setting. Here, the algorithm is fed one query at a time, and must process this query (producing $\bot$ or $\top$) before seeing the next query. The input may be adaptive—future queries may depend on the answers to earlier queries.

We focus on the interactive version; the non-interactive version can be handled similar to Report-noisy-max. We break the code into two pieces. The first piece initializes variables and computes the noisy threshold, while the second piece accepts a single new query and returns the answer.

```
proc $\mathrm{SV.I{\scriptsize NIT}}(T_{in}, \epsilon_{in})$ :
  $\epsilon \leftarrow \epsilon_{in}$;
  $T \xleftarrow{\$} \mathcal{L}_{\epsilon/2}(T_{in})$;
```

```
proc $\mathrm{SV.S{\scriptsize TEP}}(q)$ :
  $a \xleftarrow{\$} \mathcal{L}_{\epsilon/4}(\mathsf{evalQ}(q, d))$;
  if $(a < T)$ then $\{z \leftarrow \bot;\}$ else $\{z \leftarrow \top;\}$
return $z$;
```

The main procedure performs initialization, and then enters into an interactive loop between the external procedure $\mathcal{A}$—which supplies the queries—and the Sparse Vector procedure SV.Step:

```
proc $\mathrm{SV.MAIN}(Q, T, \epsilon)$ :
  $\mathrm{SV.I{\scriptsize NIT}}(T, \epsilon)$;
  $u \leftarrow 0; ans[u] \leftarrow \bot$;
  while $(u < Q)$ do
    $u \leftarrow u + 1$;
    $q[u] \leftarrow \mathcal{A}(ans[u-1])$;
    $ans[u] \leftarrow \mathrm{SV.S{\scriptsize TEP}}(q[u])$;
return $ans$;
```

Sparse Vector satisfies the following accuracy guarantee.

[2] In some presentations, the algorithm is also parameterized by the maximum number k of queries to answer. This feature is important for privacy but not accuracy, so we omit it. It is not difficult to extend the accuracy proof for answering at most k queries.

▶ **Theorem 7.** *Let $\beta \in (0,1)$. We have*

$$\vdash_\beta \mathrm{SV.}\textsc{main}(Q,T) : \top \implies \forall j \in \{1,\dots,Q\}.\ \Phi(q[j],d),\ \textit{where}$$

$$\Phi(q,d) \triangleq \left(res = \top \to \mathsf{evalQ}(q,d) > t - \frac{6}{\epsilon}\log\frac{Q+1}{\beta}\right) \\ \wedge \left(res = \bot \to \mathsf{evalQ}(q,d) < t + \frac{6}{\epsilon}\log\frac{Q+1}{\beta}\right).$$

This judgment corresponds to the accuracy guarantee for Sparse Vector from (see, e.g., Dwork and Roth [14]). Note that the error term depends logarithmically on the total number of queries Q, a key feature of Sparse Vector.

To prove this theorem, we first specify the procedures SV.Init and SV.Step. For initialization, we have

$$\vdash_{\beta/(Q+1)} \mathrm{SV.}\textsc{Init}(T,\epsilon) : \top \implies \Phi_t \qquad \text{where} \qquad \Phi_t \triangleq |t-T| < \tfrac{2}{\epsilon}\log\tfrac{Q+1}{\beta} \wedge \epsilon = \epsilon_{in}\ .$$

For the interactive step, we have

$$\vdash_{\beta/(Q+1)} \mathrm{SV.}\textsc{Step}(q) : \Phi_t \implies \Phi_t \wedge \Phi(q,d)\ .$$

Combining these two judgments, we can prove accuracy for SV.main (Theorem 7).

3.3 Online Multiplicative Weights

Our final example demonstrates how we can use the union bound to analyze a complex combination of several interactive algorithms, yielding sophisticated accuracy proofs. We will verify the *Online Multiplicative Weights* (OMW) algorithm first proposed by Hardt and Rothblum [21] and later refined by Gupta et al. [20]. Like Sparse Vector, this interactive algorithm can handle adaptive queries while guaranteeing error logarithmic in the number of queries. Unlike Sparse Vector, OMW produces approximate answers to the queries instead of just a bit representing above or below threshold.

At a high level, OMW iteratively constructs a *synthetic* version of the true database. The user can present various linear queries to the algorithm, which applies the Sparse Vector algorithm to check whether the error of the synthetic database on this query is smaller than some threshold. If so, the algorithm simply returns the approximate answer. Otherwise, it updates the synthetic database using the *multiplicative weights* update rule to better model the true database, and answers the query by adding Laplace noise to the true answer. An inductive argument shows that after enough updates, the synthetic database must be similar to the true database on *all* queries. At this point, we can answer all subsequent queries using the synthetic database alone.

In code, the following procedure implements the Online Multiplicative Weights algorithm.

```
proc $\text{MW-SV.MAIN}(d, \alpha, \epsilon, Q, X, n)$ :
  $\eta \leftarrow \alpha/2n; T \leftarrow 2\alpha; c \leftarrow 4n^2 \ln(X)/\alpha^2$;                set parameters
  $u \leftarrow 0; k \leftarrow 0; ans[k] \leftarrow \perp$;                                  initialize variables
  $mwdb \leftarrow \text{MW.INIT}(\eta, X, n); \text{SV.INIT}(T, \epsilon/4c)$;                initialize MW and SV
  while $(k < Q)$ do                                                main loop
    $k \leftarrow k + 1$;                                                  increment count of queries
    $q[k] \leftarrow \mathcal{A}(ans[k-1], mwdb)$;                                   get next query
    $approx \leftarrow \mathsf{evalQ}(q[k], mwdb)$;                                calculate approx answer
    $exact \leftarrow \mathsf{evalQ}(q[k], d)$;                                     calculate exact answer
    if $(k \geq c)$ then $ans[k] \leftarrow approx$;                            enough updates, use approx answer
    else
      $err_{>} \leftarrow \mathsf{error}(q[k], mwdb); at \leftarrow \text{SV.STEP}(err_{>})$;           check if approx answer is high
      $err_{<} \leftarrow \mathsf{invQ}(\mathsf{error}(q[k], mwdb)); bt \leftarrow \text{SV.STEP}(err_{<})$;    check if approx answer is low
      if $(at \neq \perp \vee bt \neq \perp)$ then                                large error
        $u \leftarrow u + 1$;                                              increment count of updates
        if $at \neq \perp$ then $up \leftarrow q[k]$;                                 approx answer too high
        else $up \leftarrow \mathsf{negQ}(q[k])$;                                    approx answer too low
        $mwdb \leftarrow \text{MW.STEP}(mwdb, up)$;                                update synthetic db
        $ans[k] \xleftarrow{\$} \mathcal{L}_{\epsilon/2c}(exact)$;                                estimate true answer
      else                                                          small error, do not update
        $ans[k] \leftarrow approx$;                                          answer using approx answer
return $ans$;
```

Online multiplicative weights satisfies the following accuracy guarantee.

▶ **Theorem 8.** *Let $\beta \in (0, 1)$. Then,*

$$\vdash_\beta \text{MW-SV.MAIN}(d, \alpha, \epsilon, Q, X, n) : \alpha \geq \max(\alpha_{sv}, \alpha_{lap}) \implies$$
$$\forall j.\ j \in \{1, \dots, Q\} \to |res[j] - \mathsf{evalQ}(q[j], d)| \leq \alpha,$$

$$\textit{where} \quad \gamma \triangleq 4n^2 \ln(X)/\alpha^2, \quad \alpha_{sv} \triangleq \frac{24\gamma}{\epsilon} \log \frac{2(Q+1)}{\beta}, \quad \textit{and} \quad \alpha_{lap} \triangleq \frac{4\gamma}{\epsilon} \log \frac{2\gamma}{\beta}.$$

In words, the answers to all the supplied queries are within α of the true answer if α is sufficiently large. The above judgment reflects the accuracy guarantee first proved by Hardt and Rothblum [21] and later generalized by Gupta et al. [20].

The main routine depends on the *multiplicative weights* subroutine (MW), which maintains and updates the synthetic database. Roughly, MW takes as input the current synthetic database and a query where the synthetic database gives an answer that is far from the true answer. Then, MW improves the synthetic database to better model the true database. Our implementation of MW consists of two subroutines: MW.INIT initializes the synthetic database, and MW.STEP updates the current database with a query that has high error. The code for these subroutines is somewhat technical, and we will not present it here.

Instead, we will present their specifications, which are given in terms of an expression $\Psi(x, d)$ where x is the current synthetic database and d is the true database. We omit the definition of Ψ and focus on its three key properties:

- $\Psi(x, d) \geq 0$;
- $\Psi(x, d)$ is initially bounded for the initial synthetic database; and
- $\Psi(x, d)$ decreases each time we update the synthetic database.

Functions satisfying these properties are often called *potential functions*.

The first property follows from the definition of Ψ, while the second and third properties are reflected by the specifications of the MW procedures. Concretely, we can bound the initial value of Ψ with the following specification for MW.INIT:

$$\vdash_0 \mathrm{MW.}\textsc{init}(\eta, X, n) : \top \Longrightarrow \Psi(\mathit{res}, d) \le \ln X$$

We can also show that Ψ decreases with the following specification for MW.STEP:

$$\vdash_0 \mathrm{MW.}\textsc{step}(x, q) : \top \Longrightarrow \Psi(x, d) - \Psi(\mathit{res}, d) \ge \eta(\mathsf{evalQ}(q, x) - \mathsf{evalQ}(q, d))/n - \eta^2$$

We make two remarks. First, these specifications crucially rely on the fact that q is a linear query. Second, both procedures are deterministic. For such procedures, the fragment of aHL with index $\beta = 0$ corresponds precisely to standard Hoare logic.

Now, let us briefly consider the key points in proving the main specification (Theorem 8). First, the key part of the invariant for the main loop is $\Psi(\mathit{mwdb}, d) \le \log X - u \cdot \alpha^2/4n^2$. Roughly, Ψ is initially at most $\log X$ by the specification for MW.INIT, and every time we call MW.STEP we decrease Ψ by at least $\alpha^2/4n^2$ if the update query up has error at least α. Since Ψ is always non-negative, we can find at most c queries with high error—after c updates, the synthetic database mwdb must give accurate answers on all queries.

Prior to making c updates, there are two cases for each query. If at least one of the Sparse Vector calls returns above threshold, we set the update query up to be $q[u]$ if the approximate answer is too high, otherwise we set up to be the negated query $\mathsf{neqQ}(q[u])$ if the approximate answer is too low. With this choice of update query, we can show that

$$\mathsf{evalQ}(\mathit{up}, \mathit{mwdb}) - \mathsf{evalQ}(\mathit{up}, d) \ge \alpha$$

so Ψ decreases by at least $\alpha^2/4n^2$. Then, we answer the original query $q[u]$ by adding Laplace noise, so our answer is also within α of the true answer. Otherwise, if both Sparse Vector calls return below threshold, then the query $q[u]$ is answered well by our approximation mwdb and there is no need to update mwdb or access the real database d.

The above reasoning assumes that Sparse Vector and the Laplace mechanisms are sufficiently accurate. To guarantee the former, notice that the Sparse Vector subroutine will process at most $2Q$ queries, so we assume that α is larger than the error α_{sv} guaranteed by Theorem 7 for $2Q$ queries and failure probability $\beta/2$. To guarantee the latter, notice that we sample Laplace noise at most c times—once for each update step—so we assume that α is larger than the error α_{lap} guaranteed by [LapAcc] for failure probability $\beta/2c$; by a union bound, all Laplace noises are accurate except with probability $\beta/2$. Taking $\alpha \ge \max(\alpha_{sv}, \alpha_{lap})$, both accuracy guarantees hold except with probability at most β, and we have the desired proof of accuracy for OMW (Theorem 8).

4 Related work

The semantics of probabilistic programming languages has been studied extensively since the late 70s. Kozen's seminal paper [28] studies two semantics for a core probabilistic imperative language. Other important work investigates using monads to structure the semantics of probabilistic languages; e.g. Jones and Plotkin [24]. More recent works study the semantics of probabilistic programs for applications like statistical computations [9], probabilistic inference for machine learning [10], probabilistic modeling for software defined networks [17], and more.

Likewise, deductive techniques for verifying probabilistic programs have a long history. Ramshaw [35] proposes a program logic with basic assertions of the form $\Pr[E] = p$. Hart et al. [22], Sharir et al. [39] propose a method using intermediate assertions and invariants for proving general properties of probabilistic programs. Kozen [29] introduces PPDL, a logic that can reason about expected values of general measurable functions. Morgan et al. [34] (see McIver and Morgan [31] for an extended account) propose a verification method based on computing *greatest pre-expectations*, a probabilistic analogue of Dijkstra's weakest pre-conditions. Hurd et al. [23] formalize their approach using the HOL theorem prover. Other approaches based on interactive theorem provers include the work of Audebaud and Paulin-Mohring [1], who axiomatize (discrete) probability theory and verify some examples of randomized algorithms using the Coq proof assistant. Gretz et al. [19] extend the work of Morgan et al. [34] with a formal treatment of conditioning. More recently, Rand and Zdancewic [36] formalize another Hoare logic for probabilistic programs using the Coq proof assistant. Barthe et al. [6] implement a general-purpose logic in the EasyCrypt framework, and verify a representative set of randomized algorithms. Kaminski et al. [25] develop a weakest precondition logic to reason about expected run-time of probabilistic programs.

Most of these works support general probabilistic reasoning and additional features like non-determinism, so they most likely could formalize the examples that we consider. However, our logic aHL aims at a sweet spot in the design space, combining expressivity with simplicity of the assertion language. The design of aHL is inspired by existing *relational* program logics, such as pRHL [3] and apRHL [4]. These logics support rich proofs about probabilistic properties with purely non-probabilistic assertions, using a powerful coupling abstraction from probability theory [7] rather than the union bound.

Finally, there are many algorithmic techniques for verifying probabilistic programs. Probabilistic model-checking is a successful line of research that has delivered mature and practical tools and addressed a broad range of case studies; Baier and Katoen [2], Katoen [26], Kwiatkowska et al. [30] cover some of the most interesting developments in the field. Abstract interpretation of probabilistic programs is another rich source of techniques; see e.g. Cousot and Monerau [13], Monniaux [33]. Katoen et al. [27] infer linear invariants for the pGCL language of Morgan et al. [34]. There are several approaches based on martingales for reasoning about probabilistic loops; Chakarov and Sankaranarayanan [11, 12] use martingales for inferring expectation invariants, while Ferrer Fioriti and Hermanns [16] use martingales for analyzing probabilistic termination. Sampson et al. [38] use a mix of static and dynamic analyses to check probabilistic assertions for probabilistic programs.

5 Conclusion and perspective

We propose aHL, a lightweight probabilistic Hoare logic based on the union bound. Our logic can prove properties about bad events in cryptography and accuracy of differentially private mechanisms. Of course, there are examples that we cannot verify. For instance, reasoning involving independence of random variables, a common tool when analyzing randomized algorithms, is not supported. Accordingly, a natural next step is to explore logical methods for reasoning about independence, or to embed aHL into a more general system like pGCL.

Acknowledgements This work was partially supported by NSF grants TWC-1513694, CNS-1065060 and CNS-1237235, by EPSRC grant EP/M022358/1 and by a grant from the Simons Foundation (#360368 to Justin Hsu).

References

1 P. Audebaud and C. Paulin-Mohring. Proofs of randomized algorithms in Coq. *Science of Computer Programming*, 74(8):568–589, 2009. URL `https://www.lri.fr/~paulin/ALEA/random-scp.pdf`.

2 C. Baier and J.-P. Katoen. *Principles of model checking*. MIT Press, 2008. ISBN 978-0-262-02649-9.

3 G. Barthe, B. Grégoire, and S. Zanella-Béguelin. Formal certification of code-based cryptographic proofs. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Savannah, Georgia*, pages 90–101, New York, 2009. URL `http://research.microsoft.com/pubs/185309/Zanella.2009.POPL.pdf`.

4 G. Barthe, B. Köpf, F. Olmedo, and S. Zanella-Béguelin. Probabilistic relational reasoning for differential privacy. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Philadelphia, Pennsylvania*, pages 97–110, 2012. URL `http://certicrypt.gforge.inria.fr/2012.POPL.pdf`.

5 G. Barthe, M. Gaboardi, E. J. Gallego Arias, J. Hsu, C. Kunz, and P.-Y. Strub. Proving differential privacy in Hoare logic. In *IEEE Computer Security Foundations Symposium (CSF), Vienna, Austria*, 2014. URL `http://arxiv.org/abs/1407.2988`.

6 G. Barthe, T. Espitau, M. Gaboardi, B. Grégoire, J. Hsu, and P.-Y. Strub. Formal certification of randomized algorithms. 2015. URL `http://justinh.su/files/docs/BEGGHS15paper.pdf`.

7 G. Barthe, T. Espitau, B. Grégoire, J. Hsu, L. Stefanesco, and P.-Y. Strub. Relational reasoning via probabilistic coupling. In *International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR), Suva, Fiji*, volume 9450, pages 387–401, 2015. URL `http://arxiv.org/abs/1509.03476`.

8 G. Barthe, M. Gaboardi, B. Grégoire, J. Hsu, and P.-Y. Strub. Proving differential privacy via probabilistic couplings. In *IEEE Symposium on Logic in Computer Science (LICS), New York, New York*, 2016. URL `http://arxiv.org/abs/1601.05047`. To appear.

9 S. Bhat, A. Agarwal, R. Vuduc, and A. Gray. A type theory for probability density functions. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Philadelphia, Pennsylvania*, pages 545–556, 2012. URL `http://doi.acm.org/10.1145/2103656.2103721`.

10 J. Borgström, A. D. Gordon, M. Greenberg, J. Margetson, and J. V. Gael. Measure transformer semantics for Bayesian machine learning. *Logical Methods in Computer Science*, 9(3), 2013. URL `http://dx.doi.org/10.2168/LMCS-9(3:11)2013;http://arxiv.org/abs/1308.0689`.

11 A. Chakarov and S. Sankaranarayanan. Probabilistic program analysis with martingales. In *International Conference on Computer Aided Verification (CAV), Saint Petersburg, Russia*, pages 511–526, 2013. URL `https://www.cs.colorado.edu/~srirams/papers/cav2013-martingales.pdf`.

12 A. Chakarov and S. Sankaranarayanan. Expectation invariants as fixed points of probabilistic programs. In *International Symposium on Static Analysis (SAS), Munich, Germany*, volume 8723 of *Lecture Notes in Computer Science*, pages 85–100. Springer-Verlag, 2014. URL `https://www.cs.colorado.edu/~srirams/papers/sas14-expectations.pdf`.

13 P. Cousot and M. Monerau. Probabilistic abstract interpretation. In H. Seidl, editor, *European Symposium on Programming (ESOP), Tallinn, Estonia*, volume 7211 of *Lecture Notes in Computer Science*, pages 169–193. Springer, 2012. URL `http://www.di.ens.fr/~cousot/publications.www/Cousot-Monerau-ESOP2012-extended.pdf`.

14 C. Dwork and A. Roth. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science*, 9(3-4):211–407, 2014. URL http://dx.doi.org/10.1561/0400000042.

15 C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *IACR Theory of Cryptography Conference (TCC), New York, New York*, pages 265–284, 2006. URL http://dx.doi.org/10.1007/11681878_14.

16 L. M. Ferrer Fioriti and H. Hermanns. Probabilistic termination: Soundness, completeness, and compositionality. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Mumbai, India*, pages 489–501. ACM, 2015. URL http://www.ae-info.org/attach/User/Hermanns_Holger/Publications/FH-POPL15.pdf.

17 N. Foster, D. Kozen, K. Mamouras, M. Reitblatt, and A. Silva. Probabilistic NetKAT. In *European Symposium on Programming (ESOP), Eindhoven, The Netherlands*, Lecture Notes in Computer Science, 2016.

18 M. Gaboardi, A. Haeberlen, J. Hsu, A. Narayan, and B. C. Pierce. Linear dependent types for differential privacy. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Rome, Italy*, pages 357–370, 2013. URL http://dl.acm.org/citation.cfm?id=2429113.

19 F. Gretz, J.-P. Katoen, and A. McIver. Prinsys – on a quest for probabilistic loop invariants. In *International Conference on Quantitative Evaluation of Systems (QEST)*, pages 193–208, 2013.

20 A. Gupta, A. Roth, and J. Ullman. Iterative constructions and private data release. In *IACR Theory of Cryptography Conference (TCC), Taormina, Italy*, pages 339–356, 2012. URL http://arxiv.org/abs/1107.3731.

21 M. Hardt and G. N. Rothblum. A multiplicative weights mechanism for privacy-preserving data analysis. In *IEEE Symposium on Foundations of Computer Science (FOCS), Las Vegas, Nevada*, pages 61–70, 2010. URL http://www.mit.edu/~rothblum/papers/pmw.pdf.

22 S. Hart, M. Sharir, and A. Pnueli. Termination of probabilistic concurrent programs. In *ACM Symposium on Principles of Programming Languages (POPL), Albuquerque, New Mexico*, pages 1–6, 1982. 10.1145/582153.582154. URL http://doi.acm.org/10.1145/582153.582154.

23 J. Hurd, A. McIver, and C. Morgan. Probabilistic guarded commands mechanized in HOL. *Theoretical Computer Science*, 346(1):96–112, 2005.

24 C. Jones and G. D. Plotkin. A probabilistic powerdomain of evaluations. In *IEEE Symposium on Logic in Computer Science (LICS), Asilomar, California*, pages 186–195, 1989. URL http://dx.doi.org/10.1109/LICS.1989.39173.

25 B. L. Kaminski, J.-P. Katoen, C. Matheja, and F. Olmedo. Weakest precondition reasoning for expected run-times of probabilistic programs. In *European Symposium on Programming (ESOP), Eindhoven, The Netherlands*, Lecture Notes in Computer Science, 2016.

26 J.-P. Katoen. Perspectives in probabilistic verification. In *IEEE/IFIP International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 3–10, 2008.

27 J.-P. Katoen, A. McIver, L. Meinicke, and C. Morgan. Linear-invariant generation for probabilistic programs. In R. Cousot and M. Martel, editors, *International Symposium on Static Analysis (SAS), Perpignan, France*, volume 6337 of *Lecture Notes in Computer Science*, pages 390–406. Springer, 2010.

28 D. Kozen. Semantics of probabilistic programs. In *IEEE Symposium on Foundations of Computer Science (FOCS), San Juan, Puerto Rico*, pages 101–114, 1979.

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

30 M. Z. Kwiatkowska, G. Norman, and D. Parker. Probabilistic symbolic model checking with PRISM: A hybrid approach. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Tallinn, Estonia*, pages 52–66, 2002.

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

32 F. McSherry and K. Talwar. Mechanism design via differential privacy. In *IEEE Symposium on Foundations of Computer Science (FOCS), Providence, Rhode Island*, pages 94–103, 2007. URL `http://doi.ieeecomputersociety.org/10.1109/FOCS.2007.41`.

33 D. Monniaux. Abstract interpretation of probabilistic semantics. In J. Palsberg, editor, *International Symposium on Static Analysis (SAS), Santa Barbara, California*, 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 Transactions on Programming Languages and Systems*, 18(3):325–353, 1996.

35 L. H. Ramshaw. *Formalizing the Analysis of Algorithms.* PhD thesis, Stanford University, 1979.

36 R. Rand and S. Zdancewic. VPHL: A verified partial-correctness logic for probabilistic programs. In *Mathematical Foundations of Program Semantics (MFPS)*, 2015.

37 J. Reed and B. C. Pierce. Distance makes the types grow stronger: A calculus for differential privacy. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Baltimore, Maryland*, 2010. URL `http://dl.acm.org/citation.cfm?id=1863568`.

38 A. Sampson, P. Panchekha, T. Mytkowicz, K. S. McKinley, D. Grossman, and L. Ceze. Expressing and verifying probabilistic assertions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Edinburgh, Scotland*, page 14, 2014. URL `http://research.microsoft.com/pubs/211410/passert-pldi2014.pdf`.

39 M. Sharir, A. Pnueli, and S. Hart. Verification of probabilistic programs. *SIAM Journal on Computing*, 13(2):292–314, 1984. 10.1137/0213021. URL `http://dx.doi.org/10.1137/0213021`.

This appendix details the missing definitions of this paper main body, along with the proof of soundness of the presented logic.

A A bit more on discrete distributions

We start by defining some standard sub-distributions that are needed for giving the denotation semantic of our language:

▶ **Definition 9.** Let T be some set and $x \in T$. We denote by $\mathbb{1}_x^T \in \mathbf{Distr}(T)$ (resp. $\mathbb{0}^T \in \mathbf{Distr}(T)$) the Dirac distribution over T and centered on x (resp. the null sub-distribution over T):

$$\mathbb{1}_x^T = \lambda v \in T. \begin{cases} 1 & \text{if } x = v \\ 0 & \text{otherwise} \end{cases} \qquad \mathbb{0}^T = \lambda v \in T.\, 0$$

We write $\mathbb{1}_x$ and $\mathbb{0}$, stripping T, when it is clear from the context.

Let T and U be two sets. We denote by $\int_{x \leftarrow \mu} E(x)$, where μ is a sub-distribution over T and $E : T \to \mathbf{Distr}(U)$, the sub-distribution with mass function $\lambda v.\, \sum_{x \in T} E(x)(v)\ \mu(x)$.

It is convenient to introduce the notion of restriction of a distribution.

▶ **Definition 10** (Restriction of a sub-distribution)**.** Let μ be a sub-distribution over T, and let P be a predicate over T. Then, the *restriction* of μ to P is defined as

$$\mu_{|P}(x) \triangleq \begin{cases} \mu(x) & \text{if } P(x) \\ 0 & \text{otherwise.} \end{cases}$$

From the definition, it is clear that $\Pr_{\mu_{|P}}[Q] = \Pr_\mu[P \wedge Q]$.

B Denotational semantics

We now give the denotation semantics of our language. We start by interpreting the expressions and distribution expressions, and then move to the interpretation of commands.

B.1 Types, expressions and distribution expressions

We fix a set $\mathcal{T} = \{\tau, \sigma, \ldots\}$ of types. We assume that $\mathcal{T}$ contains at least the unit type (**unit**), along with the types for booleans (**bool**) and integers (**int**). For a variable $x \in \mathbf{Vars}$, we denote the type associated to x by τ_x. Moreover, for $\tau \in \mathcal{T}$, we write $\mathbf{Vars}_\tau$ for the subset $\{x \in \mathbf{Vars} \mid \tau_x = \tau\}$ of $\mathbf{Vars}$, and require that it is infinite.

We also assume given a set $\mathcal{O}$ of operators and $\mathcal{O}_\mathcal{D}$ of distribution operators. To each operator $o \in \mathcal{O}$ is associated an arity $o : [\tau_i]_{i \le n} \to \tau$, where $[\tau_i]_i$ is the domain of o and τ its codomain. Likewise, to each distribution operator $d \in \mathcal{O}_\mathcal{D}$ is associated an arity $d : \tau \to \sigma$, meaning that d is a distribution over σ parameterized by a value of type τ.

We can now give the syntax of expressions and distribution expressions:

▶ **Definition 11** (Expressions & distribution expressions)**.** The set of expressions of type τ, written $\mathcal{E}_\tau$, is defined by:

$$\mathcal{E}_\tau ::= x \in \mathbf{Vars}_\tau \mid o(e_1, \ldots, e_n) \qquad \text{with } o : [\tau_i]_i \to \tau \text{ and } \forall i.\, e_i \in \mathcal{E}_{\tau_i}.$$

Likewise, the set of distribution expressions over σ is defined by:

$$\mathcal{D}_\tau ::= d(e) \qquad \text{with } d : \sigma \to \tau \text{ and } e \in \mathcal{E}_\sigma.$$

We now move to the interpretation of types, expressions and distribution expressions.

B.2 Interpretation of types

For any type τ, the set $[\![\tau]\!]$ denotes the interpretation of τ: $[\![\mathbf{unit}]\!] = \{\bullet\}$, $[\![\mathbf{bool}]\!] = \{\top, \bot\}$, $[\![\mathbf{int}]\!] = \mathbb{Z}$ and $[\![\mathbf{real}]\!] = \mathbb{R}$.

B.3 Interpretation of expressions

For any operator o with arity $[\tau_i]_i \to \tau$, we assume given $\overline{o} : \bigtimes_i [\![\tau_i]\!] \to [\![\tau]\!]$. The interpretation of an expression e w.r.t a *typed* valuation $\rho : \Pi(x : \mathbf{Vars}). [\![\tau_x]\!]$ (i.e. w.r.t. a function that associates a value in $[\![\tau_x]\!]$ to any variable $x \in \mathbf{Vars}$) is defined as usual: $[\![x]\!]_\rho = \rho(x)$ and $[\![o(e_1, \dots, e_n)]\!]_\rho = \overline{o}([\![e_1]\!]_\rho, \dots, [\![e_n]\!]_\rho)$. If $e \in \mathcal{E}_\tau$, we have that $[\![e]\!]_\rho \in [\![\tau]\!]$.

Likewise, for any distribution operator $d : \sigma \to \tau$ is associated a function $\overline{d}$ from $[\![\sigma]\!]$ to $\mathbf{Distr}([\![\tau]\!])$. The interpretation of a distribution expression $d(e)$ w.r.t. a valuation ρ, written $[\![d(e)]\!]_\rho$, is defined by $[\![d(e)]\!]_\rho = \overline{d}([\![e]\!]_\rho)$.

B.4 Denotational semantics

A memory m is any map of type $\Pi(x : \mathbf{Vars} \cup \{\mathfrak{a}\}). [\![\tau_x]\!]$, where $\mathfrak{a}$ is a special variable dedicated to the storage of the (shared) state of the external procedures — associating an abstract type $\tau_\mathfrak{a} = \mathfrak{A}$ to it. Note that memories can be considered as valuations, simply forgetting the binding for $\mathfrak{a}$. For any external procedure $\mathcal{A}$ taking a parameter of type τ and returning a value of type σ, we assume given an interpretation $\overline{\mathcal{A}} : \mathfrak{A} \times [\![\tau]\!] \to \mathbf{Distr}(\mathfrak{A} \times [\![\sigma]\!])$.

Finally, if f is a internal procedure, we denote by $f.\mathbf{arg}$ (resp. $f.\mathbf{body}$, $f.\mathbf{res}$) the argument name (resp. the body, the return expression) of f.

▶ **Definition 12** (Denotational Semantics). The denotational semantics of a command maps a memory to a sub-distribution over memories and is given in Figure 2, where abort is an extra command that never returns, and $(\mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ )^n_\bot$ is inductively defined by:

$$\begin{aligned} (\mathsf{if}\ e\ \mathsf{then}\ c)^0_\bot &= \mathsf{if}\ e\ \mathsf{then}\ \mathsf{abort} \\ (\mathsf{if}\ e\ \mathsf{then}\ c)^{n+1}_\bot &= \mathsf{if}\ e\ \mathsf{then}\ \{c;\ (\mathsf{if}\ e\ \mathsf{then}\ c)^n_\bot\} \end{aligned}$$

$$\begin{aligned} [\![\mathsf{abort}]\!] &= \lambda m.\, \mathbb{0} \\ [\![\mathsf{skip}]\!] &= \lambda m.\, \mathbb{1}_m \\ [\![x \leftarrow e]\!] &= \lambda m.\, \mathbb{1}_{m[x \leftarrow [\![e]\!]_m]} \\ [\![x \overset{\$}{\leftarrow} e]\!] &= \lambda m.\, \int_{v \leftarrow [\![e]\!]_m} \mathbb{1}_{m[x \leftarrow v]} \\ [\![c1;\ c2]\!] &= \lambda m.\, \int_{\xi \leftarrow [\![c_1]\!](m)} [\![c_2]\!](\xi) \\ [\![\mathsf{if}\ e\ \mathsf{then}\ c1\ \mathsf{else}\ c2]\!] &= \lambda m.\, \begin{cases} [\![c_1]\!](m) & \text{if } [\![e]\!]_m = \top \\ [\![c_2]\!](m) & \text{if } [\![e]\!]_m = \bot \end{cases} \\ [\![\mathsf{while}\ e\ \mathsf{do}\ c]\!] &= \lambda m.\, \lim_{n\infty} [\![(\mathsf{if}\ e\ \mathsf{then}\ c)^n_\bot]\!](m) \\ [\![x \leftarrow \mathcal{A}(e)]\!] &= \lambda m.\, \int_{(v_\mathfrak{a}, v) \leftarrow \overline{\mathcal{A}}(m(\mathfrak{a}), [\![e]\!]_m)} \mathbb{1}_{m[\mathfrak{a} \leftarrow v_\mathfrak{a}][x \leftarrow v]} \\ [\![x \leftarrow f(e)]\!] &= \lambda m.\, [\![f.\mathbf{arg} \leftarrow e; f.\mathbf{body}; x \leftarrow f.\mathbf{res}]\!](m) \\ &= \lambda m.\, \int_{\xi \leftarrow [\![f.\mathbf{body}]\!](m')} \mathbb{1}_{\xi[x \leftarrow \xi(f.\mathbf{res})]} \\ &\quad \text{where } m' = m[f.\mathbf{arg} \leftarrow [\![e]\!]_m] \end{aligned}$$

Figure 2 Denotational Semantics

The following lemma is useful in the proof of soundness.

▶ **Lemma 13.** *Let c be a command, m_1, m_2 be two memories s.t. $m_2 \in \mathrm{supp}(\llbracket c \rrbracket(m_1))$. Then, $\forall x \in \mathcal{X}$ that is not written by c and $m_1[x] = m_2[x]$.*

Proof. By a direct induction on the structure of c. ◀

Before moving to the soundness proof, we make clear what it means for a command to modify a variable

▶ **Definition 14.** Let c be command. The set $\mathrm{mod}(c) \subseteq \textbf{Vars}$ of program variables modifies by c is defined by induction on the structure of c:

$$\begin{aligned}
&\mathrm{mod}(\mathsf{abort} &&) = \emptyset\\
&\mathrm{mod}(\mathsf{skip} &&) = \emptyset\\
&\mathrm{mod}(x \leftarrow e &&) = \{x\}\\
&\mathrm{mod}(x \xleftarrow{\$} d(e) &&) = \{x\}\\
&\mathrm{mod}(c_1;\, c_2 &&) = \mathrm{mod}(c_1) \cup \mathrm{mod}(c_2)\\
&\mathrm{mod}(\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 &&) = \mathrm{mod}(c_1) \cup \mathrm{mod}(c_2)\\
&\mathrm{mod}(\mathsf{while}\ e\ \mathsf{do}\ c &&) = \mathrm{mod}(c)\\
&\mathrm{mod}(x \leftarrow \mathcal{A}(e) &&) = \{x\}\\
&\mathrm{mod}(x \leftarrow f(e) &&) = \{x, f.\mathbf{arg}\} \cup \mathrm{mod}(f.\mathbf{body})
\end{aligned}$$

Note that the expressions do not have side effects.

C Soundness proof

The logical entailment relation $m \models \Phi$ being abstract, we assume here the following property:

▶ **Lemma 15.** *Let m, Φ and e s.t. $m \models \Phi[e/x]$. Then, $m[x \leftarrow e] \models \Phi$.*

Note that, beside the former property, $m \models \Phi$ gives a standard model to first order connectives. We can now prove Theorem 3:

Proof of Theorem 3. The proof is by induction on the derivation of $\vdash_\beta c : \Phi \implies \Psi$, by case analysis on the last rule:

[Skip] We have $\Psi \equiv \Phi$ and $\beta = 0$. Let $m \models \Phi$. Then $\Pr_{\llbracket \mathsf{skip} \rrbracket(m)}[\neg\Psi] = \Pr_{\mathbb{1}_m}[\neg\Psi] = \sum_\xi (\mathbb{1}_m)_{|\neg\Psi}(\xi) = (\mathbb{1}_m)_{|\neg\Psi}(m) = 0$, the last equality being a direct consequence of $m \models \Phi (\equiv \Psi)$.

[Weak] We have $\vdash_{\beta'} c : \Phi' \implies \Psi'$ with i) $\models \Phi \implies \Phi'$, ii) $\models \Psi' \implies \Psi$, and iii) $\beta' \leq \beta$. Let $m \models \Phi$. By i), $m \models \Phi'$. Hence, by induction hypothesis, $\Pr_{\llbracket c \rrbracket(m)}[\neg\Psi'] \leq \beta'$. From ii) & iii), $\Pr_{\llbracket c \rrbracket(m)}[\neg\Psi] \leq \Pr_{\llbracket c \rrbracket(m)}[\neg\Psi'] \leq \beta' \leq \beta$.

[Seq] We have $\vdash_{\beta_1} c_1 : \Phi \implies \Xi$ and $\vdash_{\beta_2} c_2 : \Xi \implies \Psi$, with $c \equiv c_1;\, c_2$ and $\beta = \beta_1 + \beta_2$. Let $m \models \Phi$. Then,

$$\begin{aligned}
\Pr_{\llbracket c\rrbracket(m)}[\neg\Psi] &= \sum_{\xi\models\neg\Psi} \llbracket c_1;\, c_2\rrbracket(m)(\xi) = \sum_{\xi\models\neg\Psi}\left[\int_{\theta\leftarrow\llbracket c_1\rrbracket(m)} \llbracket c_2\rrbracket(\theta)\right](\xi)\\
&\qquad\text{(swap $\textstyle\sum$ — inner $\textstyle\sum$ being hidden in $\textstyle\int$)}\\
&= \sum_{\theta}\ \sum_{\xi\models\neg\Psi} \llbracket c_2\rrbracket(\theta)(\xi)\cdot\llbracket c_1\rrbracket(m)(\theta)\\
&\qquad\text{(split external $\textstyle\sum$ on $\models\Xi$ and refold inner Pr)}\\
&= \sum_{\theta\models\Xi}\llbracket c_1\rrbracket(m)(\theta)\cdot\underbrace{\Pr_{\llbracket c_2\rrbracket(\theta)}[\neg\Psi]}_{\leq\beta_2} + \sum_{\theta\models\neg\Xi}\llbracket c_1\rrbracket(m)(\theta)\cdot\underbrace{\Pr_{\llbracket c_2\rrbracket(\theta)}[\neg\Psi]}_{\leq 1}\\
&\leq \underbrace{\left(\sum_{\theta\models\Xi}\llbracket c_1\rrbracket(m)(\theta)\right)}_{=\Pr_{\llbracket c_1\rrbracket(m)}[\Xi]\leq 1}\cdot\beta_2 + \underbrace{\sum_{\theta\models\neg\Xi}\llbracket c_1\rrbracket(m)(\theta)}_{=\Pr_{\llbracket c_1\rrbracket(m)}[\neg\Xi]\leq\beta_1}\\
&\leq \beta_1+\beta_2
\end{aligned}$$

[Assn] We have $c \equiv x \leftarrow e$, $\phi \equiv \Xi[e/x]$, $\psi \equiv \Xi$ and $\beta = 0$. Let $m \models \Xi[e/x]$. By substitutivity of the logical entailment relation, $m[x \leftarrow \llbracket e\rrbracket_m] \models \Xi$. Then, $\Pr_{\llbracket c\rrbracket(m)}[\neg\Xi] = \sum_\xi \left[(\mathbb{1}_{m[x\leftarrow\llbracket e\rrbracket_m]})_{|\neg\Xi}\right](\xi) = \left[(\mathbb{1}_{m[x\leftarrow\llbracket e\rrbracket_m]})_{|\neg\Xi}\right](m[x\leftarrow\llbracket e\rrbracket_m]) = 0$.

[Rand] The premise directly implies the conclusion — the rule is semantical.

[If] We have $\vdash_\beta c_\top : \Phi\wedge e \implies \Psi$ and $\vdash_\beta c_\bot : \Phi\wedge\neg e \implies \Psi$, with $c \equiv \mathsf{if}\ e\ \mathsf{then}\ c_\top\ \mathsf{else}\ c_\bot$. Let $m \models \Phi$ and let $b = \llbracket e\rrbracket_m$. Then, $m \models \Phi \wedge e = b$, and by application of the induction hypothesis, $\Pr_{\llbracket c\rrbracket(m)}[\neg\Psi] = \Pr_{\llbracket c_b\rrbracket(m)}[\neg\Psi] \leq \beta$.

[While] We have $c \equiv \mathsf{while}\ e\ \mathsf{do}\ c_I$, $\Phi \equiv I \wedge e_v \leq k$, $\Psi \equiv I \wedge \neg e$ and $\beta = k\cdot\beta_I$, with $\vdash_{\beta_I} c_I : I \implies I$, $\models I \to (e_v \leq k) \wedge (e_v \leq 0 \to \neg e)$ and $\forall \eta > 0.\ \vdash_0 c_I : I \wedge e \wedge e_v = \eta \implies e_v < \eta$. The proof is done by (strong) induction on k. Let $m \models I \wedge e_v \leq k$.
If $m \models \neg e$, then $(\mathsf{if}\ e\ \mathsf{then}\ c_I\ \mathsf{else}\ )^t_\bot\ (m) = \mathbb{1}_m$ for any t. Hence, $\llbracket\mathsf{while}\ e\ \mathsf{do}\ c_I\rrbracket(m) = \lim_{n\infty}\mathbb{1}_m = \mathbb{1}_m$; and, from $m \models I \wedge \neg e$, we have $\Pr_{\llbracket c\rrbracket(m)}[\neg(I\wedge\neg e)] = \Pr_{\mathbb{1}_m}[\neg(I\wedge\neg e)] = (\mathbb{1}_m)_{|\neg(I\wedge\neg e)}\ (m) = 0 \leq k\cdot\beta_I$.
Otherwise, $m \models e$. From the logical premises, we have $m \models 0 < e_v \leq k$. Moreover, $\llbracket(\mathsf{if}\ e\ \mathsf{then}\ c_I\ \mathsf{else}\ )^{t+1}_\bot\rrbracket(m) = \llbracket c_I;\ (\mathsf{if}\ e\ \mathsf{then}\ c_I\ \mathsf{else}\ )^t_\bot\rrbracket(m)$, and thus, $\llbracket c\rrbracket(m) = \llbracket c_I;\, c\rrbracket(m)$. By a reasoning similar to the one of [Seq], using $I \wedge e_v \leq (k-1)$ as the intermediate assertion, it suffices to show that i) $\Pr_{\llbracket c_I\rrbracket(m)}[\neg(I\wedge e_v \leq (k-1))] \leq \beta_I$, and ii) for any $\xi \models I \wedge e_v \leq (k-1)$, $\Pr_{\llbracket c\rrbracket(\xi)}[\neg(I\wedge\neg e)] \leq (k-1)\cdot\beta_I$. Point ii)) is obtained by an application of the inner induction hypothesis. For Point i)), we have:

$$\Pr_{\llbracket c_I\rrbracket(m)}[\neg(I\wedge e_v \leq (k-1))] \leq \underbrace{\Pr_{\llbracket c_I\rrbracket(m)}[\neg I]}_{\leq\ \beta_I} + \underbrace{\Pr_{\llbracket c_I\rrbracket(m)}[\neg(e_v \leq (k-1))]}_{=\ 0}$$

the comparison to β_I coming form $\vdash_{\beta_I} c : I \implies I$, whereas the comparison to 0 is a direct consequence of $m \models I \wedge e \wedge e_v = k \to e_v < k$, obtained by instantiation of the logical premises.

[Ext] We have $c \equiv x \leftarrow \mathcal{A}(e)$, $\Phi \equiv \forall v.\ \Psi[v/x]$ and $\beta = 0$. Let $m \models \Phi$. Then,

$$\begin{aligned}
\Pr_{[\![c]\!](m)}[\neg\Psi] &= \sum_{\xi\models\neg\Psi}\sum_{(v_{\mathfrak{a}},v)}\left[\mathbb{1}_{\underbrace{m[\mathfrak{a}\leftarrow v_{\mathfrak{a}}][x\leftarrow v]}_{\alpha(v_{\mathfrak{a}},v)}}\cdot\overline{\mathcal{A}}(m[\mathfrak{a}],[\![e]\!]_m)(v_{\mathfrak{a}},v)\right](\xi)\\
&= \sum_{(v_{\mathfrak{a}},v)}\sum_{\xi}\left[\mathbb{1}_{\alpha(v_{\mathfrak{a}},v)}\cdot\underbrace{\overline{\mathcal{A}}(m[\mathfrak{a}],[\![e]\!]_m)(v_{\mathfrak{a}},v)}_{\leq 1}\right]_{|\neg\Psi}(\xi)\\
&\leq \sum_{(v_{\mathfrak{a}},v)}\sum_{\xi}\left[\mathbb{1}_{\alpha(v_{\mathfrak{a}},v)}\right]_{|\neg\Psi}(\xi) = \sum_{(v_{\mathfrak{a}},v)}\underbrace{\left[\mathbb{1}_{\alpha(v_{\mathfrak{a}},v)}\right]_{|\neg\Psi}(\alpha(v_{\mathfrak{a}},v))}_{=\,0}\\
&= 0
\end{aligned}$$

noticing that $m\models\forall v.\,\Psi[v/x]$ with $\mathfrak{a}\notin\Psi$ implies $\alpha(v_{\mathfrak{a}},v)\models\Psi$.

[Call] This is a direct consequence of the properties for [Seq] & [Assn].

[Frame] We have $\Phi\equiv\Psi$, $\beta=0$ and c does not modify the variables free in Φ. Then,

$$\Pr_{[\![c]\!](m)}[\neg\Psi] = \Pr_{[\![c]\!](m)}[\neg\Phi] = \sum_{\xi\models\neg\Phi}[\![c]\!](m)(\xi) = \sum_{\xi\in\mathrm{supp}([\![c]\!](m))}([\![c]\!](m))_{|\neg\Phi}(\xi).$$

From Lemma 13, for any $\xi\in\mathrm{supp}([\![c]\!](m))$, considering that c does not modify the free variables $\mathrm{FV}(\Phi)$ of Φ, we have $m_{|\,\mathrm{FV}(\phi)}=\xi_{|\,\mathrm{FV}(\phi)}$. Hence, $m\models\Phi$ implies $\xi\models\Phi$, and:

$$\sum_{\xi\in\mathrm{supp}([\![c]\!](m))}\underbrace{([\![c]\!](m))_{|\neg\Phi}(\xi)}_{=\,0}=0.$$

[And] We have $\vdash_{\beta_1} c:\Phi\implies\Psi_1$ and $\vdash_{\beta_2} c:\Phi\implies\Psi_2$ with $\Psi\equiv\Psi_1\wedge\Psi_2$ and $\beta=\beta_1+\beta_2$. Let $m\models\Phi$. Then, by induction hypothesis, $\Pr_{[\![c]\!](m)}[\neg\Phi_1]\leq\beta_1$ and $\Pr_{[\![c]\!](m)}[\neg\Phi_2]\leq\beta_2$. Hence, $\Pr_{[\![c]\!](m)}[\neg(\Phi_1\wedge\Phi_2)]=\Pr_{[\![c]\!](m)}[\neg\Phi_1]+\Pr_{[\![c]\!](m)}[\neg\Phi_2]-\Pr_{[\![c]\!](m)}[\neg\Phi_1\wedge\neg\Phi_2]\leq\Pr_{[\![c]\!](m)}[\neg\Phi_1]+\Pr_{[\![c]\!](m)}[\neg\Phi_2]\leq\beta_1+\beta_2=\beta$.

[Or] We have $\vdash_{\beta} c:\Phi_1\implies\Psi$ and $\vdash_{\beta} c:\Phi_2\implies\Psi$ with $\Phi\equiv\Phi_1\vee\Phi_2$. Let $m\models\Phi$. Then, $m\models\Phi_1$ or $m\models\Phi_2$. W.l.o.g. we can assume $m\models\Phi_1$. We then obtain the expected result $\Pr_{[\![c]\!](m)}[\neg\Psi]\leq\beta$ by application of the induction hypothesis.

[False] We have $\Psi\equiv\bot$ and $\beta=1$. Let $m\models\Phi$. Then $\Pr_{[\![c]\!](m)}[\neg\Psi]\leq 1=\beta$. ◀