Constraint-Based Synthesis of Coupling Proofs*

Aws Albarghouthi[1] and Justin Hsu[2,3]

[1] University of Wisconsin–Madison, Madison, WI
[2] University College London, London, UK
[3] Cornell University, Ithaca, NY

Abstract. *Proof by coupling* is a classical technique for proving properties about pairs of randomized algorithms by carefully *relating* (or *coupling*) two probabilistic executions. In this paper, we show how to automatically construct such proofs for probabilistic programs. First, we present *f-coupled postconditions*, an abstraction describing two correlated program executions. Second, we show how properties of f-coupled postconditions can imply various probabilistic properties of the original programs. Third, we demonstrate how to reduce the proof-search problem to a purely logical *synthesis problem* of the form $\exists f. \forall X. \varphi$, making probabilistic reasoning unnecessary. We develop a prototype implementation to automatically build coupling proofs for probabilistic properties, including uniformity and independence of program expressions.

1 Introduction

In this paper, we aim to automatically synthesize *coupling proofs* for probabilistic programs and properties. Originally designed for proving properties comparing two probabilistic programs—so-called *relational properties*—a coupling proof describes how to correlate two executions of the given programs, simulating both programs with a single probabilistic program. By reasoning about this combined, *coupled* process, we can often give simpler proofs of probabilistic properties for the original pair of programs.

A number of recent works have leveraged this idea to verify relational properties of randomized algorithms, including differential privacy [12, 10, 8], security of cryptographic protocols [9], convergence of Markov chains [11], robustness of machine learning algorithms [7], and more. Recently, Barthe et al. [6] showed how to reduce certain *non-relational* properties—which describe a single probabilistic program—to relational properties of two programs, by duplicating the original program or by sequentially composing it with itself.

While coupling proofs can simplify reasoning about probabilistic properties, they are not so easy to use; most existing proofs are carried out manually in relational program logics using interactive theorem provers. In a nutshell, the main challenge in a coupling proof is to select a correlation for each pair of corresponding sampling instructions, aiming to induce a particular relation between

* The full version of this paper is available at `https://arxiv.org/abs/1804.04052`.

the outputs of the coupled process; this relation then implies the desired relational property. Just like finding inductive invariants in proofs for deterministic programs, picking suitable couplings in proofs can require substantial ingenuity.

To ease this task, we recently showed how to cast the search for coupling proofs as a program synthesis problem [1], giving a way to automatically find sophisticated proofs of differential privacy previously beyond the reach of automated verification. In the present paper, we build on this idea and present a general technique for constructing coupling proofs, targeting *uniformity* and *probabilistic independence* properties. Both are fundamental properties in the analysis of randomized algorithms, either in their own right or as prerequisites to proving more sophisticated guarantees; uniformity states that a randomized expression takes on all values in a finite range with equal probability, while probabilistic independence states that two probabilistic expressions are somehow uncorrelated—learning the value of one reveals no additional information about the value of the other.

Our techniques are inspired by the automated proofs of differential privacy we considered previously [1], but the present setting raises new technical challenges.

Non-lockstep execution. To prove differential privacy, the behavior of a single program is compared on two related inputs. To take advantage of the identical program structure, previous work restricted attention to *synchronizing* proofs, where the two executions can be analyzed assuming they follow the same control flow path. In contrast, coupling proofs for uniformity and independence often require relating two programs with different shapes, possibly following completely different control flows [6].

To overcome this challenge, we take a different approach. Instead of incrementally finding couplings for corresponding pairs of sampling instructions—requiring the executions to be tightly synchronized—we first lift all sampling instructions to the front of the program and pick a coupling once and for all. The remaining execution of both programs can then be encoded separately, with no need for lockstep synchronization (at least for loop-free programs—looping programs require a more careful treatment).

Richer space of couplings. The heart of a coupling proof is selecting—among multiple possible options—a particular correlation for each pair of random sampling instructions. Random sampling in differentially private programs typically use highly domain-specific distributions, like the Laplace distribution, which support a small number of useful couplings. Our prior work leveraged this feature to encode a collection of primitive couplings into the synthesis system. However, this is no longer possible when programs sample from distributions supporting richer couplings, like the uniform distribution. Since our approach coalesces all sampling instructions at the beginning of the program (more generally, at the head of the loop), we also need to find couplings for products of distributions.

We address this problem in two ways. First, we allow couplings of two sampling instructions to be specified by an injective function f from one range to an-other. Then, we impose requirements—encoded as standard logical constraints—to ensure that f indeed represents a coupling; we call such couplings *f-couplings*.

More general class of properties. Finally, we consider a broad class of properties rather than just differential privacy. While we focus on uniformity and independence for concreteness, our approach can establish general equalities between products of probabilities, i.e., probabilistic properties of the form

$$\prod_{i=1}^{m} \Pr[e_i \in E_i] = \prod_{j=1}^{n} \Pr[e'_j \in E'_j],$$

where e_i and e'_j are program expressions in the first and second programs respectively, and E_i and E'_j are predicates. As an example, we automatically establish a key step in the proof of Bertrand's Ballot theorem [20].

Paper Outline. After overviewing our technique on a motivating example (Section 2), we detail our main contributions.

- **Proof technique:** We introduce *f-coupled postconditions*, a form of postcondition for two probabilistic programs where random sampling instructions in the two programs are correlated by a function f. Using f-coupled postconditions, we present proof rules for establishing uniformity and independence of program variables, fundamental properties in the analysis of randomized algorithms (Section 3).
- **Reduction to constraint-based synthesis:** We demonstrate how to automatically find coupling proofs by transforming our proof rules into logical constraints of the form $\exists f. \forall X. \varphi$—a synthesis problem. A satisfiable constraint shows the existence of a function f—essentially, a compact encoding of a coupling proof—implying the target property (Section 4).
- **Extension to looping programs:** We extend our technique to reason about loops, by requiring synchronization at the loop head and finding a coupled invariant (Section 5).
- **Implementation and evaluation:** We implement our technique and evaluate it on several case studies, automatically constructing coupling proofs for interesting properties of a variety of algorithms (Section 6).

We conclude by comparing our technique with related approaches (Section 7).

2 Overview and Illustration

2.1 Introducing f-Couplings

A Simple Example. We begin by illustrating f-couplings over two identical Bernoulli distributions, denoted by the following *probability mass functions*: $\mu_1(x) = \mu_2(x) = 0.5$ for all $x \in \mathbb{B}$ (where $\mathbb{B} = \{\mathit{true}, \mathit{false}\}$). In other words, the distribution μ_i returns *true* with probability 0.5, and *false* with probability 0.5.

An *f-coupling* for μ_1, μ_2 is a function $f : \mathbb{B} \to \mathbb{B}$ from the domain of the first distribution ($\mathbb{B}$) to the domain of the second (also $\mathbb{B}$); f should be injective and satisfy the *monotonicity property*: $\mu_1(x) \leqslant \mu_2(f(x))$ for all $x \in \mathbb{B}$. In other words, f relates each element $x \in \mathbb{B}$ with an element $f(x)$ that has an equal or larger probability in μ_2. For example, consider the function $f_\neg$ defined as

$$f_\neg(x) = \neg x.$$

This function relates *true* in μ_1 with *false* in μ_2, and vice versa. Observe that $\mu_1(x) \leqslant \mu_2(f_\neg(x))$ for all $x \in \mathbb{B}$, satisfying the definition of an $f_\neg$-coupling. We write $\mu_1 \leftrightsquigarrow^{f_\neg} \mu_2$ when there is an $f_\neg$-coupling for μ_1 and μ_2.

Using f-Couplings. An f-coupling can imply useful properties about the distributions μ_1 and μ_2. For example, suppose we want to prove that $\mu_1(\mathit{true}) = \mu_2(\mathit{false})$. The fact that there is an $f_\neg$-coupling of μ_1 and μ_2 immediately implies the equality: by the monotonicity property,

$$\mu_1(\mathit{true}) \leqslant \mu_2(f_\neg(\mathit{true})) = \mu_2(\mathit{false})$$
$$\mu_1(\mathit{false}) \leqslant \mu_2(f_\neg(\mathit{false})) = \mu_2(\mathit{true})$$

and therefore $\mu_1(\mathit{true}) = \mu_2(\mathit{false})$. More generally, it suffices to find an f-coupling of μ_1 and μ_2 such that

$$\underbrace{\{(x, f(x)) \mid x \in \mathbb{B}\}}_{\Psi_f} \subseteq \{(z_1, z_2) \mid z_1 = \mathit{true} \iff z_2 = \mathit{false}\},$$

where Ψ_f is induced by f; in particular, the $f_\neg$-coupling satisfies this property.

2.2 Simulating a Fair Coin

Now, let's use f-couplings to prove more interesting properties. Consider the program fairCoin in Figure 1; the program simulates a fair coin by flipping a possibly biased coin that returns *true* with probability $p \in (0, 1)$, where p is a program parameter. Our goal is to prove that for any p, the output of the program is a uniform distribution—it simulates a fair coin. We consider two separate copies of fairCoin generating distributions μ_1 and μ_2 over the returned value x for the same bias p, and we construct a coupling showing $\mu_1(\mathit{true}) = \mu_2(\mathit{false})$, that is, heads and tails have equal probability.

```
fun fairCoin($p \in (0,1)$)
    $x \leftarrow \mathit{false}$
    $y \leftarrow \mathit{false}$
    while $x = y$ do
        $x \sim$ bern($p$)
        $y \sim$ bern($p$)
    return $x$
```

Fig. 1: Simulating a fair coin using an unfair one

Constructing f-Couplings. At first glance, it is unclear how to construct an f-coupling; unlike the distributions in our simple example, we do not have a concrete description of μ_1 and μ_2 as uniform distributions (indeed, this is what we are trying to establish). The key insight is that we do not need to construct our coupling in one shot. Instead, we can specify a coupling for the concrete, primitive sampling instructions in the body of the loop—which we know sample from bern(p)—and then extend to a f-coupling for the whole loop and μ_1, μ_2.

For each copy of $\mathsf{fairCoin}$, we coalesce the two sampling statements inside the loop into a single sampling statement from the product distribution:

$$x, y \sim \mathsf{bern}(p) \times \mathsf{bern}(p)$$

We have two such joint distributions $\mathsf{bern}(p) \times \mathsf{bern}(p)$ to couple, one from each copy of $\mathsf{fairCoin}$. We use the following function $f_{swap} : \mathbb{B}^2 \to \mathbb{B}^2$:

$$f_{swap}(x, y) = (y, x)$$

which exchanges the values of x and y. Since this is an injective function satisfying the monotonicity property

$$(\mathsf{bern}(p) \times \mathsf{bern}(p))(x, y) \leqslant (\mathsf{bern}(p) \times \mathsf{bern}(p))(f_{swap}(x, y))$$

for all $(x, y) \in \mathbb{B} \times \mathbb{B}$ and $p \in (0, 1)$, we have an f_{swap}-coupling for the two copies of $\mathsf{bern}(p) \times \mathsf{bern}(p)$.

Analyzing the Loop. To extend a f_{body}-coupling on loop bodies to the entire loop, it suffices to check a synchronization condition: the coupling from f_{body} must ensure that the loop guards are equal so the two executions synchronize at the loop head. This holds in our case: every time the first program executes the statement $x, y \sim \mathsf{bern}(p) \times \mathsf{bern}(p)$, we can think of x, y as non-deterministically set to some values (a, b), and the corresponding variables in the second program as set to $f_{swap}(a, b) = (b, a)$. The loop guards in the two programs are equivalent under this choice, since $a = b$ is equivalent to $b = a$, hence we can analyze the loops in lockstep. In general, couplings enable us to relate samples from a pair of probabilistic assignments as if they were selected non-deterministically, often avoiding quantitative reasoning about probabilities.

Our constructed coupling for the loop guarantees that (i) both programs exit the loop at the same time, and (ii) when the two programs exit the loop, x takes opposite values in the two programs. In other words, there is an f_{loop}-coupling of μ_1 and μ_2 for some function f_{loop} such that

$$\Psi_{f_{loop}} \subseteq \{(z_1, z_2) \mid z_1 = \mathit{true} \iff z_2 = \mathit{false}\}, \tag{1}$$

implying $\mu_1(\mathit{true}) = \mu_2(\mathit{false})$. Since both distributions are output distributions of $\mathsf{fairCoin}$—hence $\mu_1 = \mu_2$—we conclude that $\mathsf{fairCoin}$ simulates a fair coin.

Note that our approach does not need to construct f_{loop} concretely—this function may be highly complex. Instead, we only need to show that $\Psi_{f_{loop}}$ (or some over-approximation) lies inside the target relation in Formula 1.

Achieving Automation. Observe that once we have fixed an f_{body}-coupling for the sampling instructions inside the loop body, checking that the f_{loop}-coupling satisfies the conditions for uniformity (Formula 1) is essentially a program verification problem. Therefore, we can cast the problem of constructing a coupling proof as a logical problem of the form $\exists f. \forall X. \varphi$, where f is the f-coupling we need to discover and $\forall X. \varphi$ is a constraint ensuring that (i) f indeed represents an f-coupling, and (ii) the f-coupling implies uniformity. Thus, we can use established synthesis-verification techniques to solve the resulting constraints (see, e.g., [27, 2, 13]).

3 A Proof Rule for Coupling Proofs

In this section, we develop a technique for constructing couplings and formalize proof rules for establishing uniformity and independence properties over program variables. We begin with background on probability distributions and couplings.

3.1 Distributions and Couplings

Distributions. A function $\mu : B \to [0,1]$ defines a *distribution* over a countable set B if $\sum_{b \in B} \mu(b) = 1$. We will often write $\mu(A)$ for a subset $A \subseteq B$ to mean $\sum_{x \in A} \mu(x)$. We write $\mathit{dist}(B)$ for the set of all distributions over B.

We will need a few standard constructions on distributions. First, the *support* of a distribution μ is defined as $\mathit{supp}(\mu) = \{b \in B \mid \mu(b) > 0\}$. Second, for a distribution on pairs $\mu \in \mathit{dist}(B_1 \times B_2)$, the first and second *marginals* of μ, denoted $\pi_1(\mu)$ and $\pi_2(\mu)$ respectively, are distributions over B_1 and B_2:

$$\pi_1(\mu)(b_1) \triangleq \sum_{b_2 \in B_2} \mu(b_1, b_2) \qquad\qquad \pi_2(\mu)(b_2) \triangleq \sum_{b_1 \in B_1} \mu(b_1, b_2).$$

Couplings. Let $\Psi \subseteq B_1 \times B_2$ be a binary relation. A *Ψ-coupling* for distributions μ_1 and μ_2 over B_1 and B_2 is a distribution $\mu \in \mathit{dist}(B_1 \times B_2)$ with (*i*) $\pi_1(\mu) = \mu_1$ and $\pi_2(\mu) = \mu_2$; and (*ii*) $\mathit{supp}(\mu) \subseteq \Psi$. We write $\mu_1 \leftrightsquigarrow^{\Psi} \mu_2$ when there exists a Ψ-coupling between μ_1 and μ_2.

An important fact is that an injective function $f : B_1 \to B_2$ where $\mu_1(b) \leqslant \mu_2(f(b))$ for all $b \in B_1$ induces a coupling between μ_1 and μ_2; this follows from a general theorem by Strassen [28], see also [23]. We write $\mu_1 \leftrightsquigarrow^{f} \mu_2$ for $\mu_1 \leftrightsquigarrow^{\Psi_f} \mu_2$, where $\Psi_f = \{(b_1, f(b_1)) \mid b_1 \in B_1\}$. The existence of a coupling can imply various useful properties about the two distributions. The following general fact will be the most important for our purposes—couplings can prove equalities between probabilities.

Proposition 1. *Let $E_1 \subseteq B_1$ and $E_2 \subseteq B_2$ be two events, and let $\Psi_= \triangleq \{(b_1, b_2) \mid b_1 \in E_1 \iff b_2 \in E_2\}$. If $\mu_1 \leftrightsquigarrow^{\Psi_=} \mu_2$, then $\mu_1(E_1) = \mu_2(E_2)$.*

3.2 Program Model

Our program model uses an imperative language with probabilistic assignments, where we can draw a random value from primitive distributions. We consider the easier case of loop-free programs first; we consider looping programs in Section 5.

Syntax. A (loop-free) program P is defined using the following grammar:

$$\begin{aligned} P \coloneqq\ & V \leftarrow \mathit{exp} && \text{(assignment)} \\ & \mid V \sim \mathit{dexp} && \text{(probabilistic assignment)} \\ & \mid \texttt{if}\ \mathit{bexp}\ \texttt{then}\ P\ \texttt{else}\ P && \text{(conditional)} \\ & \mid P; P && \text{(sequential composition)} \end{aligned}$$

where V is the set of variables that can appear in P, *exp* is an expression over V, and *bexp* is a Boolean expression over V. A probabilistic assignment samples from a probability distribution defined by expression *dexp*; for instance, *dexp* might be $\mathsf{bern}(p)$, the Bernoulli distribution with probability p of returning *true*. We use $V^I \subseteq V$ to denote the set of input program variables, which are never assigned to. All other variables are assumed to be defined before use.

We make a few simplifying assumptions. First, distribution expressions only mention input variables V^I, e.g., in the example above, $\mathsf{bern}(p)$, we have $p \in V^I$. Also, all programs are in *static single assignment* (SSA) form, where each variable is assigned to only once and are well-typed. These assumptions are relatively minor; they can can be verified using existing tools, or lifted entirely at the cost of slightly more complexity in our encoding.

Semantics. A state s of a program P is a valuation of all of its variables, represented as a map from variables to values, e.g., $s(x)$ is the value of $x \in V$ in s. We extend this mapping to expressions: $s(exp)$ is the valuation of *exp* in s, and $s(dexp)$ is the probability distribution defined by *dexp* in s.

We use S to denote the set of all possible program states. As is standard [24], we can give a semantics of P as a function $[\![P]\!] : S \to dist(S)$ from states to distributions over states. For an output distribution $\mu = [\![P]\!](s)$, we will abuse notation and write, e.g., $\mu(x = y)$ to denote the probability of the event that the program returns a state s where $s(x = y) = true$.

Self-Composition. We will sometimes need to simulate two separate executions of a program with a single probabilistic program. Given a program P, we use P_i to denote a program identical to P but with all variables *tagged* with the subscript i. We can then define the *self-composition*: given a program P, the program $P_1; P_2$ first executes P_1, and then executes the (separate) copy P_2.

3.3 Coupled Postconditions

We are now ready to present the f*-coupled postcondition*, an operator for approximating the outputs of two coupled programs.

Strongest Postcondition. We begin by defining a standard strongest postcondition operator over single programs, treating probabilistic assignments as no-ops. Given a set of states $Q \subseteq S$, we define post as follows:

$$\begin{aligned}
\mathsf{post}(v \leftarrow exp, Q) &= \{s[v \mapsto s(exp)] \mid s \in Q\} \\
\mathsf{post}(v \sim dexp, Q) &= Q \\
\mathsf{post}(\texttt{if } bexp \texttt{ then } P \texttt{ else } P', Q) &= \{s' \mid s \in Q, s' \in \mathsf{post}(P, s), s(bexp) = true\} \\
&\quad \cup \{s' \mid s \in Q, s' \in \mathsf{post}(P', s), s(bexp) = false\} \\
\mathsf{post}(P; P', Q) &= \mathsf{post}(P', \mathsf{post}(P, Q))
\end{aligned}$$

where $s[v \mapsto c]$ is state s with variable v mapped to the value c.

f**-Coupled Postcondition.** We rewrite programs so that all probabilistic assignments are combined into a single probabilistic assignment to a vector of variables appearing at the beginning of the program, i.e., an assignment of the form $\boldsymbol{v} \sim \mathit{dexp}$ in P and $\boldsymbol{v}' \sim \mathit{dexp}'$ in P', where $\boldsymbol{v}, \boldsymbol{v}'$ are vectors of variables. For instance, we can combine $x \sim \mathsf{bern}(0.5); y \sim \mathsf{bern}(0.5)$ into the single statement $x, y \sim \mathsf{bern}(0.5) \times \mathsf{bern}(0.5)$.

Let B, B' be the domains of $\boldsymbol{v}$ and $\boldsymbol{v}'$, $f : B \to B'$ be a function, and $Q \subseteq S \times S'$ be a set of pairs of input states, where S and S' are the states of P and P', respectively. We define the f-coupled postcondition operator cpost as

$$\begin{aligned} \mathsf{cpost}(P, P', Q, f) &= \{(\mathsf{post}(P, s), \mathsf{post}(P', s')) \mid (s, s') \in Q'\} \\ &\quad \text{where } Q' = \{(s[\boldsymbol{v} \mapsto \boldsymbol{b}], s'[\boldsymbol{v}' \mapsto f(\boldsymbol{b})]) \mid (s, s') \in Q, \boldsymbol{b} \in B\}, \\ \text{assuming that} &\quad \forall (s, s') \in Q.\, s(\mathit{dexp}) \leftrightsquigarrow^{f} s'(\mathit{dexp}'). \end{aligned} \tag{2}$$

The intuition is that the values drawn from sampling assignments in both programs are coupled using the function f. Note that this operation non-deterministically assigns $\boldsymbol{v}$ from P with some values $\boldsymbol{b}$, and $\boldsymbol{v}'$ with $f(\boldsymbol{b})$. Then, the operation simulates the executions of the two programs. Formula 2 states that there is an f-coupling for every instantiation of the two distributions used in probabilistic assignments in both programs.

Example 1. Consider the simple program P defined as $x \sim \mathsf{bern}(0.5); x = \neg x$ and let $f_\neg(x) = \neg x$. Then, $\mathsf{cpost}(P, P, Q, f_\neg)$ is $\{(s, s') \mid s(x) = \neg s'(x)\}$.

The main soundness theorem shows there is a probabilistic coupling of the output distributions with support contained in the coupled postcondition (we defer all proofs to the full version of this paper).

Theorem 1. *Let programs P and P' be of the form $\boldsymbol{v} \sim \mathit{dexp}; P_D$ and $\boldsymbol{v}' \sim \mathit{dexp}'; P'_D$, for deterministic programs P_D, P'_D. Given a function $f : B \to B'$ satisfying Formula 2, for every $(s, s') \in S \times S'$ we have $[\![P]\!](s) \leftrightsquigarrow^{\Psi} [\![P']\!](s')$, where $\Psi = \mathsf{cpost}(P, P', (s, s'), f)$.*

3.4 Proof Rules for Uniformity and Independence

We are now ready to demonstrate how to establish uniformity and independence of program variables using f-coupled postconditions. We will continue to assume that random sampling commands have been lifted to the front of each program, and that f satisfies Formula 2.

Uniformity. Consider a program P and a variable $v^* \in V$ of finite, non-empty domain B. Let $\mu = [\![P]\!](s)$ for some state $s \in S$. We say that variable v^* is *uniformly distributed* in μ if $\mu(v^* = b) = \frac{1}{|B|}$ for every $b \in B$.

The following theorem connects uniformity with f-coupled postconditions.

Theorem 2 (Uniformity). *Consider a program P with $\boldsymbol{v} \sim \mathit{dexp}$ as its first statement and a designated return variable $v^* \in V$ with domain B. Let $Q = \{(s, s) \mid s \in S\}$ be the input relation. If we have*

$$\exists f.\, \mathsf{cpost}(P, P, Q, f) \subseteq \{(s, s') \in S \times S \mid s(v^*) = b \iff s'(v^*) = b'\}$$

for all $b, b' \in B$, then for any input $s \in S$ the final value of v^* is uniformly distributed over B in $[\![P]\!](s)$.

The intuition is that in the two f-coupled copies of P, the first v^* is equal to b exactly when the second v^* is equal to b'. Hence, the probability of returning b in the first copy and b' in the second copy are the same. Repeating for every pair of values b, b', we conclude that v^* is uniformly distributed.

Example 2. Recall Example 1 and let $b = \mathit{true}$ and $b' = \mathit{false}$. We have

$$\mathsf{cpost}(P, P, Q, f_\neg) \subseteq \{(s, s') \in S \times S \mid s(x) = b \iff s'(x) = b'\}.$$

This is sufficient to prove uniformity (the case with $b = b'$ is trivial).

Independence. We now present a proof rule for independence. Consider a program P and two variables $v^*, w^* \in V$ with domains B and B', respectively. Let $\mu = [\![P]\!](s)$ for some state $s \in S$. We say that v^*, w^* are *probabilistically independent* in μ if $\mu(v^* = b \wedge w^* = b') = \mu(v^* = b) \cdot \mu(w^* = b')$ for every $b \in B$ and $b' \in B'$.

The following theorem connects independence with f-coupled postconditions. We will self-compose two tagged copies of P, called P_1 and P_2.

Theorem 3 (Independence). *Assume a program P and define the relation*

$$Q = \{(s, s_1 \oplus s_2) \mid s \in S, s_i \in S_i, s(v) = s_i(v_i), \textit{for all } v \in V^I\},$$

where $\oplus$ takes the union of two maps with disjoint domains. Fix some $w^*, v^* \in V$ with domains B, B', and assume that for all $b \in B$, $b' \in B'$, there exists a function f such that $\mathsf{cpost}(P, (P_1; P_2), Q, f)$ is contained in

$$\{(s', s'_1 \oplus s'_2) \mid s'(v^*) = b \wedge s'(w^*) = b' \iff s'_1(v^*_1) = b \wedge s'_2(w^*_2) = b'\}.$$

Then, w^*, v^* are independently distributed in $[\![P]\!](s)$ for all inputs $s \in S$.

The idea is that under the coupling, the probability of P returning $v^* = b \wedge w^* = b'$ is the same as the probability of P_1 returning $v^* = b$ and P_2 returning $w^* = b'$, for all values b, b'. Since P_1 and P_2 are two independent executions of P by construction, this establishes independence of v^* and w^*.

4 Constraint-Based Formulation of Proof Rules

In Section 3, we formalized the problem of constructing a coupling proof using f-coupled postconditions. We now automatically find such proofs by posing the problem as a constraint, where a solution gives a function f establishing our desired property.

4.1 Generating Logical and Probabilistic Constraints

Logical Encoding. We first encode program executions as formulas in first-order logic, using the following encoding function:

$$\begin{aligned}
\mathsf{enc}(v \leftarrow \mathit{exp}) &\triangleq v = \mathit{exp}\\
\mathsf{enc}(v \sim \mathit{dexp}) &\triangleq \mathit{true}\\
\mathsf{enc}(\texttt{if } \mathit{bexp} \texttt{ then } P \texttt{ else } P') &\triangleq (\mathit{bexp} \Rightarrow \mathsf{enc}(P)) \wedge (\neg \mathit{bexp} \Rightarrow \mathsf{enc}(P'))\\
\mathsf{enc}(P; P') &\triangleq \mathsf{enc}(P) \wedge \mathsf{enc}(P')
\end{aligned}$$

We assume a direct correspondence between expressions in our language and the first-order theory used for our encoding, e.g., linear arithmetic. Note that the encoding disregards probabilistic assignments, encoding them as *true*; this mimics the semantics of our strongest postcondition operator post. Probabilistic assignments will be handled via a separate encoding of f-couplings.

As expected, enc reflects the strongest postcondition post.

Lemma 1. *Let P be a program and let ρ be any assignment of the variables. An assignment ρ' agreeing with ρ on all input variables V^I satisfies the constraint* $\mathsf{enc}(P)[\rho'/V]$ *precisely when* $\mathsf{post}(P, \{\rho\}) = \{\rho'\}$, *treating ρ, ρ' as program states.*

Uniformity Constraints. We can encode the conditions in Theorem 2 for showing uniformity as a logical constraint. For a program P and a copy P_1, with first statements $\boldsymbol{v} \sim \mathit{dexp}$ and $\boldsymbol{v}_1 \sim \mathit{dexp}_1$, we define the constraints:

$$\boxed{\begin{aligned}
&\forall a, a'. \exists f. \forall V, V_1.\\
&\qquad (V^I = V_1^I \wedge \boldsymbol{v}_1 = f(\boldsymbol{v}) \wedge \mathsf{enc}(P) \wedge \mathsf{enc}(P_1)) &\quad (3)\\
&\qquad\qquad \Longrightarrow (v^* = a \iff v_1^* = a')\\
&\qquad V^I = V_1^I \Longrightarrow \mathit{dexp} \leftrightsquigarrow^f \mathit{dexp}_1 &\quad (4)
\end{aligned}}$$

Note that this is a second-order formula, as it quantifies over the *uninterpreted function* f. The left side of the implication in Formula 3 encodes an f-coupled execution of P and P_1, starting from equal initial states. The right side of this implication encodes the conditions for uniformity, as in Theorem 2.

Formula 4 ensures that there is an f-coupling between dexp and dexp_1 for any initial state; recall that dexp may mention input variables V^I. The constraint $\mathit{dexp} \leftrightsquigarrow^f \mathit{dexp}_1$ is not a standard logical constraint—intuitively, it is satisfied if $\mathit{dexp} \leftrightsquigarrow^f \mathit{dexp}_1$ holds for some interpretation of f, dexp, and dexp_1.

Example 3. The constraint

$$\exists f. \forall p, p'. p = p' \Rightarrow \mathsf{bern}(p) \leftrightsquigarrow^f \mathsf{bern}(p')$$

holds by setting f to the identity function id, since for any $p = p'$ we have an f-coupling $\mathsf{bern}(p) \leftrightsquigarrow^{\mathrm{id}} \mathsf{bern}(p')$.

Example 4. Consider the program $x \sim \mathsf{bern}(0.5); y = \neg x$. The constraints for uniformity of y are

$$\forall a, a'.\, \exists f.\, \forall V, V_1.(x_1 = f(x) \land y = \neg x \land y_1 = \neg x_1) \implies (y = a \iff y_1 = a')$$
$$\mathsf{bern}(0.5) \leftrightsquigarrow^{f} \mathsf{bern}(0.5).$$

Since there are no input variables, $V^I = V_1^I$ is equivalent to *true*.

Theorem 4 (Uniformity constraints). *Fix a program P and variable $v^* \in V$. Let φ be the uniformity constraints in Formulas 3 and 4. If φ is valid, then v^* is uniformly distributed in $[\![P]\!](s)$ for all $s \in S$.*

Independence Constraints. Similarly, we can characterize independence constraints using the conditions in Theorem 3. After transforming the program $P_1; P_2$ to start with the single probabilistic assignment statement $\boldsymbol{v}_{1,2} \sim \mathit{dexp}_{1,2}$, combining probabilistic assignments in P_1 and P_2, we define the constraints:

$$\begin{aligned}&\forall a, a'.\exists f.\, \forall V, V_1, V_2.\\ &\qquad (V^I = V_1^I = V_2^I \land \boldsymbol{v}_{1,2} = f(\boldsymbol{v}) \land \mathsf{enc}(P) \land \mathsf{enc}(P_1; P_2))\\ &\qquad\qquad \implies (v^* = a \land w^* = a' \iff v_1^* = a \land w_2^* = a') \qquad (5)\\ &\qquad V^I = V_1^I = V_2^I \implies \mathit{dexp} \leftrightsquigarrow^{f} \mathit{dexp}_{1,2} \qquad (6)\end{aligned}$$

Theorem 5 (Independence constraints). *Fix a program P and two variables $v^*, w^* \in V$. Let φ be the independence constraints from Formulas 5 and 6. If φ is valid, then v^*, w^* are independent in $[\![P]\!](s)$ for all $s \in S$.*

4.2 Constraint Transformation

To solve our constraints, we transform our constraints into the form $\exists f.\, \forall X.\, \varphi$, where φ is a first-order formula. Such formulas can be viewed as *synthesis problems*, and are often solvable automatically using standard techniques.

We perform our transformation in two steps. First, we transform our constraint into the form $\exists f.\, \forall X.\, \varphi_p$, where φ_p still contains the coupling constraint. Then, we replace the coupling constraint with a first-order formula by logically encoding primitive distributions as uninterpreted functions.

Quantifier Reordering. Our constraints are of the form $\forall a, a'.\, \exists f.\, \forall X.\, \varphi$. Intuitively, this means that for *every* possible value of a, a', we want *one* function f satisfying $\forall X.\, \varphi$. We can pull the existential quantifier $\exists f$ to the outermost level by extending the function with additional parameters for a, a', thus defining a different function for every interpretation of a, a'. For the uniformity constraints this transformation yields the following formulas:

$$\begin{aligned}&\exists g.\, \forall a, a'.\, \forall V, V_1.\\ &\qquad (V^I = V_1^I \land \boldsymbol{v}_1 = g(a, a', \boldsymbol{v}) \land \mathsf{enc}(P) \land \mathsf{enc}(P_1)) \qquad (7)\\ &\qquad\qquad \implies (v^* = a \iff v_1^* = a')\\ &\qquad V^I = V_1^I \implies \mathit{dexp} \leftrightsquigarrow^{g(a,a',-)} \mathit{dexp}_1 \qquad (8)\end{aligned}$$

where $g(a, a', -)$ is the function after partially applying g.

Transforming Coupling Constraints. Our next step is to eliminate coupling constraints. To do so, we use the definition of f-coupling, which states that $\mu_1 \leftrightsquigarrow^f \mu_2$ if (i) f is injective and (ii) $\forall x.\, \mu_1(x) \leqslant \mu_2(f(x))$. The first constraint (injectivity) is straightforward. For the second point (monotonicity), we can encode distribution expressions—which represent functions to reals—as uninterpreted functions, which we then further constrain. For instance, the coupling constraint $\mathsf{bern}(p) \leftrightsquigarrow^f \mathsf{bern}(p')$ can be encoded as

$$\begin{aligned}
&\forall x, y.\, x \neq y \Rightarrow f(x) \neq f(y) && \text{(injectivity)}\\
&\quad \forall x.\, h(x) \leqslant h'(f(x)) && \text{(monotonicity)}\\
&\quad \forall x.\, \mathit{ite}(x = \mathit{true}, h(x) = p, h(x) = 1 - p) && (\mathsf{bern}(p) \text{ encoding})\\
&\quad \forall x.\, \mathit{ite}(x = \mathit{true}, h'(x) = p', h'(x) = 1 - p') && (\mathsf{bern}(p') \text{ encoding})
\end{aligned}$$

where $h, h' : \mathbb{B} \to \mathbb{R}^{\geqslant 0}$ are uninterpreted functions representing the probability mass functions of $\mathsf{bern}(p)$ and $\mathsf{bern}(p')$; note that the third constraint encodes the distribution $\mathsf{bern}(p)$, which returns *true* with probability p and false with probability $1 - p$, and the fourth constraint encodes $\mathsf{bern}(p')$.

Note that if we cannot encode the definition of the distribution in our first-order theory (e.g., if it requires non-linear constraints), or if we do not have a concrete description of the distribution, we can simply elide the last two constraints and under-constrain h and h'. In Section 6 we use this feature to prove properties of a program encoding a Bayesian network, where the primitive distributions are unknown program parameters.

Theorem 6 (Transformation soundness). *Let φ be the constraints generated for some program P. Let φ' be the result of applying the above transformations to φ. If φ' is valid, then φ is valid.*

Constraint Solving. After performing these transformations, we finally arrive at constraints of the form $\exists g.\, \forall a, a'.\, \forall V.\, \varphi$, where φ is a first-order formula. These exactly match constraint-based program synthesis problems. In Section 6, we use SMT solvers and enumerative synthesis to handle these constraints.

5 Dealing with Loops

So far, we have only considered loop-free programs. In this section, we our approach to programs with loops.

***f*-Coupled Postconditions and Loops.** We consider programs of the form

$$\mathsf{while}\ \mathit{bexp}\ P^b$$

where P^b is a loop-free program that begins with the statement $\boldsymbol{v} \sim \mathit{dexp}$; our technique can also be extended to handle nested loops. We assume all programs terminate with probability 1 for any initial state; there are numerous systems for verifying this basic property automatically (see, e.g., [15–17]). To extend our f-coupled postconditions, we let $\mathsf{cpost}(P, P', Q, f)$ be the smallest set I satisfying:

$$Q \subseteq I \qquad \text{(initiation)}$$
$$\mathsf{cpost}(P^b, P^{b\prime}, I_{en}, f) \subseteq I \qquad \text{(consecution)}$$
$$I \subseteq \{s(\mathit{bexp}) = s'(\mathit{bexp}') \mid s \in S, s' \in S'\} \qquad \text{(synchronization)}$$

where $I_{en} \triangleq \{(s, s') \in I \mid s(\mathit{bexp}) = \mathit{true}\}$.

Intuitively, the set I is the least inductive invariant for the two coupled programs running with synchronized loops. Theorem 1, which establishes that f-coupled postconditions result in couplings over output distributions, naturally extends to a setting with loops.

Constraint Generation. To prove uniformity, we generate constraints much like the loop-free case except that we capture the invariant I, modeled as a relation over the variables of both programs, using a *Constrained Horn-Clause* (CHC) encoding. As is standard, we use V', V_1' to denote primed copies of program variables denoting their value after executing the body, and we assume that $\mathsf{enc}(P^b)$ encodes a loop-free program as a transition relation from states over V to states over V'.

$$\forall a, a'. \exists f, I. \forall V, V_1, V', V_1'.$$
$$V^I = V_1^I \implies I(V, V_1) \qquad \text{(initiation)}$$
$$I(V, V_1) \wedge \mathit{bexp} \wedge \boldsymbol{v}_1' = f(\boldsymbol{v}') \wedge \mathsf{enc}(P^b) \wedge \mathsf{enc}(P_1^b) \implies I(V', V_1') \qquad \text{(consecution)}$$
$$I(V, V_1) \implies \mathit{bexp} = \mathit{bexp}_1 \qquad \text{(synchronization)}$$
$$I(V, V_1) \implies \mathit{dexp} \leftrightsquigarrow^f \mathit{dexp}_1 \qquad \text{(coupling)}$$
$$I(V, V_1) \wedge \neg \mathit{bexp} \implies (v^* = a \iff v_1^* = a') \qquad \text{(uniformity)}$$

The first three constraints encode the definition of cpost; the last two ensure that f constructs a coupling and that the invariant implies the uniformity condition when the loop terminates. Using the technique presented in Section 4.2, we can transform these constraints into the form $\exists f, I. \forall X. \varphi$. That is, in addition to discovering the function f, we need to discover the invariant I.

Proving independence in looping programs poses additional challenges, as directly applying the self-composition construction from Section 3 requires relating a single loop with two loops. When the number of loop iterations is deterministic, however, we may simulate two sequentially composed loops with a single loop that interleaves the iterations (known as *synchronized* or *cross* product [29, 4]) so that we reduce the synthesis problem to finding a coupling for two loops.

6 Implementation and Evaluation

We now discuss our implementation and five case studies used for evaluation.

```
fun fairCoin($p \in (0,1)$)
    $x \leftarrow false$
    $y \leftarrow false$
    while $x = y$ do
        $x \sim$ bern($p$)
        $y \sim$ bern($p$)
    return $x$

fun fairDie
    $x \leftarrow false$
    $y \leftarrow false$
    $z \leftarrow false$
    while $x = y = z$ do
        $x \sim$ bern(0.5)
        $y \sim$ bern(0.5)
        $z \sim$ bern(0.5)
    return $(x, y, z)$
```

```
fun noisySum($n, p \in (0,1)$)
    $sum \leftarrow 0$
    for $i = 1, \ldots, n$ do
        $noise[i] \sim$ bern($p$)
        $sum \leftarrow sum + noise[i]$
    return $sum$

fun bayes($\mu, \mu', \mu''$)
    $x \sim \mu$
    $y \sim \mu'$
    $z \sim \mu''$
    $w \leftarrow f(x, y)$
    $w' \leftarrow g(y, z)$
    return $(w, w')$
```

```
fun ballot($n$)
    $tie \leftarrow false$
    $x_A \leftarrow 0$
    $x_B \leftarrow 0$
    for $i = 1, \ldots, n$ do
        $r \sim$ bern(0.5)
        if $r = 0$ then
            $x_A \leftarrow x_A + 1$
        else
            $x_B \leftarrow x_B + 1$
        if $i = 1$ then
            $first \leftarrow r$
        if $x_A = x_B$ then
            $tie \leftarrow true$
    return $(first, tie)$
```

Fig. 2: Case study programs

Implementation. To solve formulas of the form $\exists f.\, \forall X.\, \varphi$, we implemented a simple solver using a *guess-and-check* loop: We iterate through various interpretations of f, insert them into the formula, and check whether the resulting formula is valid. In the simplest case, we are searching for a function f from n-tuples to n-tuples. For instance, in Section 2.2, we discovered the function $f(x, y) = (y, x)$. Our implementation is parameterized by a grammar defining an infinite set of interpretations of f, which involves permuting the arguments (as above), conditionals, and other basic operations (e.g., negation for Boolean variables). For checking validity of $\forall X.\, \varphi$ given f, we use the Z3 SMT solver [19] for loop-free programs. For loops, we use an existing constrained-Horn-clause solver based on the MathSAT SMT solver [18].

Benchmarks and Results. As a set of case studies for our approach, we use 5 different programs collected from the literature and presented in Figure 2. For these programs, we prove uniformity, (conditional) independence properties, and other probabilistic equalities. For instance, we use our implementation to prove a main lemma for the Ballot theorem [20], encoded as the program ballot.

Figure 3 shows the time and number of loop iterations required by our implementation to discover a coupling proof. The small number of iterations and time needed demonstrates the simplicity of the discovered proofs. For instance, the ballot theorem was proved in 3 seconds and only 4 iterations, while the fairCoin example (illustrated in Section 2.2) required only two iterations and 1.4 seconds. In all cases, the size of the synthesize function f in terms of depth of its AST is no more than 4. We describe these programs and properties in a bit more detail.

Case Studies: Uniformity (fairCoin, fairDie). The first two programs produce uniformly random values. Our approach synthesizes a coupling proof certifying uniformity for both of these programs. The first program fairCoin, which we saw in Section 2.2, produces a fair coin flip given access to biased coin flips by repeatedly flipping two coins while they are equal, and returning the result of the first coin as soon as the flips differ. Note that the bias of the coin flip is a program parameter, and not fixed statically. The synthesized coupling swaps the result of the two samples, mapping the values of (x, y) to (y, x).

The second program fairDie gives a different construction for simulating a roll of a fair die given fair coin flips. Three fair coins are repeatedly flipped as long as they are all equal; the returned triple is the binary representation of a number in $\{1, \dots, 6\}$, the result of the simulated roll. The synthesized coupling is a bijection on triples of booleans $\mathbb{B} \times \mathbb{B} \times \mathbb{B}$; fixing any two possible output triples (b_1, b_2, b_3) and (b'_1, b'_2, b'_3) of distinct booleans, the coupling maps $(b_1, b_2, b_3) \mapsto (b'_1, b'_2, b'_3)$ and vice versa, leaving all other triples unchanged.

Program	Iters.	Time(s)
fairCoin	2	1.4
fairDie	9	6.1
noisySum	4	0.2
bayes	5	0.4
ballot	4	3.0

Fig. 3: Statistics

Case Studies: Independence (noisySum, bayes). In the next two programs, our approach synthesizes coupling proofs of independence and conditional independence of program variables in the output distribution. The first program, noisySum, is a stylized program inspired from privacy-preserving algorithms that sum a series of noisy samples; for giving accuracy guarantees, it is often important to show that the noisy draws are probabilistically independent. We show that any pair of samples are independent.

The second program, bayes, models a simple Bayesian network with three independent variables x, y, z and two dependent variables w and w', computed from (x, y) and (y, z) respectively. We want to show that w and w' are independent conditioned on any value of y; intuitively, w and w' only depend on each other through the value of y, and are independent otherwise. We use a constraint encoding similar to the encoding for showing independence to find a coupling proof of this fact. Note that the distributions μ, μ', μ'' of x, y, z are unknown parameters, and the functions f and g are also uninterpreted. This illustrates the advantage of using a constraint-based technique—we can encode unknown distributions and operations as uninterpreted functions.

Case Studies: Probabilistic Equalities (ballot). As we mentioned in Section 1, our approach extends naturally to proving general probabilistic equalities beyond uniformity and independence. To illustrate, we consider a lemma used to prove Bertrand's Ballot theorem [20]. Roughly speaking, this theorem considers counting ballots one-by-one in an election where there are n_A votes cast for candidate A and n_B votes cast for candidate B, where n_A, n_B are parameters. If $n_A > n_B$ (so A is the winner) and votes are counted in a uniformly random order, the Ballot theorem states that the probability that A leads throughout the whole counting process—without any ties—is precisely $(n_A - n_B)/(n_A + n_B)$.

One way of proving this theorem, sometimes called André's reflection principle, is to show that the probability of counting the first vote for A and reaching a tie is equal to the probability of counting the first vote for B and reaching a tie. We simulate the counting process slightly differently—instead of drawing a uniform order to count the votes, our program draws uniform samples for votes—but the original target property is equivalent to the equality

$$\Pr[\mathit{first}_1 = 0 \land \mathit{tie}_1 \land \psi(x_{A1}, x_{B1})] = \Pr[\mathit{first}_2 = 1 \land \mathit{tie}_2 \land \psi(x_{A2}, x_{B2})] \quad (9)$$

with $\psi(x_{Ai}, x_{Bi})$ is $x_{Ai} = n_A \wedge x_{Bi} = n_B$. Our approach synthesizes a coupling and loop invariant showing that the coupled post-condition is contained in

$$\{(s_1, s_2) \mid s_1(\mathit{first} = 0 \wedge \mathit{tie} \wedge \psi(x_A, x_B)) \iff s_2(\mathit{first} = 0 \wedge \mathit{tie} \wedge \psi(x_A, x_B))\},$$

giving Formula (9) by Proposition 1 (see Barthe et al. [6] for more details).

7 Related Work

Probabilistic programs have been a long-standing target of formal verification. We compare with two of the most well-developed lines of research: probabilistic model checking and deductive verification via program logics or expectations.

Probabilistic Model Checking. Model checking has proven to be a powerful tool for verifying probabilistic programs, capable of automated proofs for various probabilistic properties (typically encoded in probabilistic temporal logics); there are now numerous mature implementations (see, e.g., [21] or [3, Ch. 10] for more details). In comparison, our approach has the advantage of being fully constraint-based. This gives it a number of unique features: (i) it applies to programs with unknown inputs and variables over infinite domains; (ii) it applies to programs sampling from distributions with parameters, or even ones sampling from unknown distributions modeled as uninterpreted functions in first-order logic; (iii) it applies to distributions over infinite domains; and (iv) the generated coupling proofs are compact. At the same time, our approach is specialized to the coupling proof technique and is likely to be more incomplete.

Deductive Verification. Compared to general deductive verification systems for probabilistic programs, like program logics [26, 5, 22, 14] or techniques reasoning by pre-expectations [25], the main benefit of our technique is automation—deductive verification typically requires an interactive theorem prover to manipulate complex probabilistic invariants. In general, the coupling proof method limits reasoning about probabilities and distributions to just the random sampling commands; in the rest of the program, the proof can avoid quantitative reasoning entirely. As a result, our system can work with non-probabilistic invariants and achieve full automation. Our approach also smoothly handles properties involving the probabilities of multiple events, like probabilistic independence, unlike techniques that analyze probabilistic events one-by-one.

Acknowledgements. We thank Samuel Drews, Calvin Smith, and the anonymous reviewers for their helpful comments. Justin Hsu was partially supported by ERC grant #679127 and NSF grant #1637532. Aws Albarghouthi was supported by NSF grants #1566015, #1704117, and #1652140.

References

1. Albarghouthi, A., Hsu, J.: Synthesizing coupling proofs of differential privacy. Proceedings of the ACM on Programming Languages 2(POPL), 58:1–58:30 (2018), `http://doi.acm.org/10.1145/3158146`

2. Alur, R., Bodik, R., Juniwal, G., Martin, M.M., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: Formal Methods in Computer-Aided Design (FMCAD), Portland, Oregon. pp. 1–8. IEEE (2013)
3. Baier, C., Katoen, J.P., Larsen, K.G.: Principles of model checking. MIT Press (2008)
4. Barthe, G., Crespo, J.M., Kunz, C.: Relational verification using product programs. In: International Symposium on Formal Methods (FM), Limerick, Ireland. Lecture Notes in Computer Science, vol. 6664, pp. 200–214. Springer-Verlag (2011), https://software.imdea.org/~ckunz/rellog/long-rellog.pdf
5. Barthe, G., Espitau, T., Gaboardi, M., Grégoire, B., Hsu, J., Strub, P.Y.: A program logic for probabilistic programs. In: European Symposium on Programming (ESOP), Thessaloniki, Greece (2018), https://justinh.su/files/papers/ellora.pdf, to appear.
6. Barthe, G., Espitau, T., Grégoire, B., Hsu, J., Strub, P.Y.: Proving uniformity and independence by self-composition and coupling. In: International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR), Maun, Botswana. EPiC Series in Computing, vol. 46, pp. 385–403 (2017), https://arxiv.org/abs/1701.06477
7. Barthe, G., Espitau, T., Grégoire, B., Hsu, J., Strub, P.: Proving expected sensitivity of probabilistic programs. Proceedings of the ACM on Programming Languages 2(POPL), 57:1–57:29 (2018), http://doi.acm.org/10.1145/3158145
8. Barthe, G., Fong, N., Gaboardi, M., Grégoire, B., Hsu, J., Strub, P.Y.: Advanced probabilistic couplings for differential privacy. In: ACM SIGSAC Conference on Computer and Communications Security (CCS), Vienna, Austria (2016), https://arxiv.org/abs/1606.07143
9. Barthe, G., Fournet, C., Grégoire, B., Strub, P.Y., Swamy, N., Zanella-Béguelin, S.: Probabilistic relational verification for cryptographic implementations. In: ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), San Diego, California. pp. 193–206 (2014), https://research.microsoft.com/en-us/um/people/nswamy/papers/rfstar.pdf
10. Barthe, G., Gaboardi, M., Grégoire, B., Hsu, J., Strub, P.Y.: Proving differential privacy via probabilistic couplings. In: IEEE Symposium on Logic in Computer Science (LICS), New York, New York. pp. 749–758 (2016), http://arxiv.org/abs/1601.05047
11. Barthe, G., Grégoire, B., Hsu, J., Strub, P.Y.: Coupling proofs are probabilistic product programs. In: ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Paris, France. pp. 161–174 (2017), http://arxiv.org/abs/1607.03455
12. Barthe, G., Köpf, B., Olmedo, F., Zanella-Béguelin, S.: Probabilistic relational reasoning for differential privacy. ACM Transactions on Programming Languages and Systems 35(3), 9 (2013), http://software.imdea.org/~bkoepf/papers/toplas13.pdf
13. Beyene, T., Chaudhuri, S., Popeea, C., Rybalchenko, A.: A constraint-based approach to solving games on infinite graphs. In: ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), San Diego, California. pp. 221–233 (2014)
14. Chadha, R., Cruz-Filipe, L., Mateus, P., Sernadas, A.: Reasoning about probabilistic sequential programs. Theoretical Computer Science 379(1), 142–165 (2007)
15. Chatterjee, K., Fu, H., Goharshady, A.K.: Termination analysis of probabilistic programs through Positivstellensatz's. In: International Conference on Computer Aided Verification (CAV), Toronto, Ontario. Lecture Notes in Computer

Science, vol. 9779, pp. 3–22. Springer-Verlag (2016), https://doi.org/10.1007/978-3-319-41528-4_1

16. Chatterjee, K., Fu, H., Novotný, P., Hasheminezhad, R.: Algorithmic analysis of qualitative and quantitative termination problems for affine probabilistic programs. In: ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Saint Petersburg, Florida. pp. 327–342 (2016), https://doi.acm.org/10.1145/2837614.2837639
17. Chatterjee, K., Novotný, P., Žikelić, Đ.: Stochastic invariants for probabilistic termination. In: ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Paris, France. pp. 145–160 (2017), https://doi.acm.org/10.1145/3009837.3009873
18. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Rome, Italy. Lecture Notes in Computer Science, vol. 7795, pp. 93–107. Springer-Verlag (2013)
19. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Budapest, Hungary. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer-Verlag (2008)
20. Feller, W.: An Introduction to Probability Theory and Its Applications, vol. 1. Wiley, third edn. (1968)
21. Forejt, V., Kwiatkowska, M., Norman, G., Parker, D.: Automated verification techniques for probabilistic systems. In: International School on Formal Methods for the Design of Computer, Communication and Software Systems. pp. 53–113. Springer (2011)
22. den Hartog, J.: Probabilistic extensions of semantical models. Ph.D. thesis, Vrije Universiteit Amsterdam (2002)
23. Hsu, J.: Probabilistic Couplings for Probabilistic Reasoning. Ph.D. thesis, University of Pennsylvania (2017), https://arxiv.org/abs/1710.09951
24. Kozen, D.: Semantics of probabilistic programs. Journal of Computer and System Sciences 22(3), 328–350 (1981), https://www.sciencedirect.com/science/article/pii/0022000081900362
25. Morgan, C., McIver, A., Seidel, K.: Probabilistic predicate transformers. ACM Transactions on Programming Languages and Systems 18(3), 325–353 (1996), dl.acm.org/ft_gateway.cfm?id=229547
26. Rand, R., Zdancewic, S.: VPHL: A verified partial-correctness logic for probabilistic programs. In: Conference on the Mathematical Foundations of Programming Semantics (MFPS), Nijmegen, The Netherlands (2015)
27. Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: International Conference on Architectural Support for Programming Langauages and Operating Systems (ASPLOS), San Jose, California. pp. 404–415 (2006), http://doi.acm.org/10.1145/1168857.1168907
28. Strassen, V.: The existence of probability measures with given marginals. The Annals of Mathematical Statistics pp. 423–439 (1965), https://projecteuclid.org/euclid.aoms/1177700153
29. Zaks, A., Pnueli, A.: CoVaC: Compiler validation by program analysis of the cross-product. In: International Symposium on Formal Methods (FM), Turku, Finland. Lecture Notes in Computer Science, vol. 5014, pp. 35–51. Springer-Verlag (2008), https://llvm.org/pubs/2008-05-CoVaC.pdf