Relational reasoning via probabilistic coupling

Gilles Barthe[1], Thomas Espitau[1,2], Benjamin Grégoire[3], Justin Hsu[4], Léo Stefanesco[1,5], and Pierre-Yves Strub[1]

[1] IMDEA Software [2] ENS Cachan [3] Inria
[4] University of Pennsylvania [5] ENS Lyon

Abstract. Probabilistic coupling is a powerful tool for analyzing probabilistic processes. Roughly, coupling two processes requires finding an appropriate witness process that characterizes both processes in the same probability space. Applications of coupling include reasoning about convergence of distributions, and *stochastic dominance*—a probabilistic version of a monotonicity property.
While the mathematical definition of coupling looks rather complex and difficult to manipulate, we show that the relational program logic pRHL—the logic underlying the EasyCrypt cryptographic proof assistant—internalizes a generalization of probabilistic coupling. We demonstrate how to express and verify classic examples of couplings in pRHL, and we mechanically verifying several couplings in EasyCrypt.

1 Introduction

Probabilistic couplings [9, 7, 10] are a powerful mathematical tool for reasoning about pairs of *probabilistic processes*: streams of values that evolve randomly according to some rule. While the two processes may be difficult to analyze independently, a probabilistic coupling arranges processes $\{u_i\}$, $\{v_i\}$ in the same space—for the simplest form of couplings, by viewing the pair of processes as randomly evolving *pairs* of values $\{(u_i, v_i)\}$—while ensuring certain coupling requirements. In this way, a coupling can coordinate the samples between the two processes so that the coupled process satisfies certain properties.

From the point of view of program verification, a coupling is a *relational* program property, which describes the relation between two programs (perhaps one program run on two different inputs, or two completely different programs). At first, it may seem that coupling is merely a single example of a relational property, but coupling is a particularly interesting property for several reasons.

Useful consequences. Couplings are known to imply many other relational properties, and are often used as a tool in mathematical proofs.

A classic use of coupling is showing that the distribution of the value of two random processes started in different locations eventually converges to the same distribution if we run the processes long enough. This property is a kind of *memorylessness*—or *Markovian*—property: The long-term behavior of the process is independent of where it started from. To prove memorylessness, the

typical strategy is to couple the two processes so that their values move closer together; once the values meet, the two processes move together, yielding the same distribution.

A different use of couplings is showing that one (numeric-valued) process is, in some sense, bigger than the other. This statement has to be interpreted carefully—since both processes evolve independently, we can't guarantee that one process is always larger than the other on all traces. *Stochastic domination* turns out to be the right definition: for any k, we require $\Pr[u \geq k] > \Pr[v \geq k]$. To verify this property, it is sufficient to find a coupling of a particular form.

Relational from non-relational. In many examples, the behavior of the second coupled process is specified by the behavior of the first; in such cases, the coupling allows us to reason just about the first process. In other words, a coupling allows us to prove certain relational properties as a property of a single program.

Compositional proofs. Not only is coupling a useful property, proofs of coupling are also intriguing. Typically, couplings are proved by considering corresponding samples of the two processes, step by step; paper proofs call this process "building a coupling", reflecting the piecewise construction of the proof. In this sense, couplings can be proved locally by considering small pieces of the programs in isolation. This feature greatly simplifies mechanical verification of couplings.

Contributions

In this paper, we apply relational program verification to probabilistic couplings. While the mathematical definition of coupling is rather involved and seemingly far from program verification technology, our primary insight is that the logic pRHL from Barthe, Grégoire, and Zanella Béguelin [2] already internalizes coupling in disguise. More precisely, pRHL is built around a *lifting* construction, which turns a relation R on two sets A and B into a relation $R^\dagger$ over the set of sub-distributions over A and the set of sub-distributions over B. Two programs are related by $R^\dagger$ precisely when there exists a coupling of their output sub-distributions whose support only contains pairs of values (u, v) which satisfy R.

This observation has three immediate consequences. First, by selecting the relation R appropriately, we can model a wide variety of coupling properties, like distribution equivalence and stochastic domination. Second, by utilizing the proof system of pRHL, we have a convenient method for constructing couplings by reasoning about the programs while abstracting away the underlying details of the coupling. Finally, we can leverage EasyCrypt, a proof assistant implementing an extended version of pRHL, to mechanical verify couplings.

2 Preliminaries

Probabilistic coupling. We begin by giving an overview of probabilistic coupling. As we described before, a coupling places two probabilistic processes (viewed as probability distributions) in the same probabilistic space.

We will work with sub-distributions over discrete (finite or countable) sets. A sub-distribution μ over a discrete set A is a function $A \to [0,1]$ such that $\sum_{a \in A} \mu(a) \leq 1$, and its support $\mathsf{supp}(\mu)$ is the pre-image of $(0,1]$. We let $\mathbf{Distr}(A)$ denote the set of sub-distributions over A. Every sub-distribution can be given a monadic structure; the unit operator maps every element a in the underlying set to its Dirac distribution δ_a and the monadic composition $\mathsf{bind}(\mu, F) \in \mathbf{Distr}(B)$ of $\mu \in \mathbf{Distr}(A)$ and $F : A \to \mathbf{Distr}(B)$ is $\mathsf{bind}(\mu, F)(b) = \sum_{a \in A} \mu(a) \times F(a)(b)$.

When working with sub-distributions over tuples, the probabilistic versions of the usual projections on tuples are called *marginals*. The first and second marginals $\pi_1(\mu)$ and $\pi_2(\mu)$ of a distribution μ over $A \times B$ are defined by $\pi_1(\mu)(a) = \sum_{b \in B} \mu(a,b)$ and $\pi_2(\mu)(b) = \sum_{a \in A} \mu(a,b)$.

We can now formally define coupling.

Definition 1. *The* Frechet class $\mathfrak{F}(\mu_1, \mu_2)$ *of two sub-distributions* μ_1 *and* μ_2 *over* A *and* B *respectively is the set of sub-distributions* μ *over* $A \times B$ *such that* $\pi_1(\mu) = \mu_1$ *and* $\pi_2(\mu) = \mu_2$*. Two distributions* μ_1, μ_2 *are said to be* coupled *with witness* μ *if* $\mu \in \mathfrak{F}(\mu_1, \mu_2)$*, i.e.* μ *is in the Frechet class of* μ_1, μ_2*.*

Lifting relations. Before introducing pRHL, we describe the *lifting* construction. This operation allows pRHL to make statements about pairs of (sub-)distributions, and is a generalized form of probabilistic coupling.

The idea is to define a family of couplings based on the support of the witness distribution. Given a relation $R \subseteq A \times B$ and two distributions μ_1 and μ_2 over A and B respectively, we let $\mathfrak{L}_R(\mu_1, \mu_2)$ denote the subset of sub-distributions $\mu \in \mathfrak{F}(\mu_1, \mu_2)$ such that $\mathsf{supp}(\mu) \subseteq R$. Given a ground relation R, we view distributions in $\mathfrak{L}_R$ as witnesses for a *lifted* relation on distributions.

Definition 2. *The* lifting *of a relation* $R \subseteq A \times B$ *is the relation* $R^\dagger \subseteq \mathbf{Distr}(A) \times \mathbf{Distr}(B)$ *with* $\mu_1 \; R^\dagger \; \mu_2$ *iff* $\mathfrak{L}_R(\mu_1, \mu_2) \neq \emptyset$*.*

Before turning to the definition of pRHL, we give some intuition for why lifting is useful. Roughly, if we know two distributions are related by a lifted relation $R^\dagger$, we can treat two samples from the distribution as if they were related by R. In other words, the lifting machinery gives a powerful way to translate between information about distributions and information about samples. Deng and Du [6] provide an excellent introductory exposition to lifting, and give several equivalent characterizations of lifting, including an inductive definition and a definition based on maximal flows.

2.1 A pRHL primer

We are now ready to present pRHL, a relational program logic for probabilistic computations. In its original form [2], implemented in the EasyCrypt proof assistant [4], pRHL reasons about programs written in a imperative language extended with random assignments with the following syntax of commands:

$$c ::= x \leftarrow e \mid x \xleftarrow{\$} d \mid \mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c \mid \mathsf{while}\ e\ \mathsf{do}\ c \mid \mathsf{skip} \mid c;\ c$$

$$\textsc{Sample}\ \frac{f \in T_1 \xrightarrow{1-1} T_2 \qquad \forall v \in T_1.\ d_1(v) = d_2(f\ v)}{\vDash x_1 \xleftarrow{\$} d_1 \sim x_2 \xleftarrow{\$} d_2 : \forall v, \Phi[v/x_1, f(v)/x_2] \Rightarrow \Phi}$$

$$\textsc{If}\ \frac{\Psi \Rightarrow e_1 = e_2 \qquad \vDash c_1 \sim c_2 : \Psi \wedge e_1 \Rightarrow \Phi \qquad \vDash c_1' \sim c_2' : \Psi \wedge \neg e_1 \Rightarrow \Phi}{\vDash \mathsf{if}\ e_1\ \mathsf{then}\ c_1\ \mathsf{else}\ c_1' \sim \mathsf{if}\ e_2\ \mathsf{then}\ c_2\ \mathsf{else}\ c_2' : \Psi \Rightarrow \Phi}$$

$$\textsc{While}\ \frac{\Phi \Rightarrow e_1 = e_2 \qquad \vDash c_1 \sim c_2 : \Phi \wedge e_1 \Rightarrow \Phi}{\vDash \mathsf{while}\ e_1\ \mathsf{do}\ c_1 \sim \mathsf{while}\ e_2\ \mathsf{do}\ c_2 : \Phi \Rightarrow \Phi \wedge \neg e_1}$$

Fig. 1: Two-sided proof rules (selection)

where e ranges over expressions, d ranges over distribution expressions, and $x \xleftarrow{\$} d$ stores a sample from d into x. Commands are interpreted as functions from memories to distributions over memories; using the fixed point theorem for Banach spaces, one can define for each command c a function $[\![c]\!] : \mathsf{Mem} \to \mathbf{Distr}(\mathsf{Mem})$, where Mem is the set of well-typed maps from program variables to values.

Assertions in the language are first-order formulae over generalized expressions. The latter are built from tagged variables x_1 and x_2, which correspond to the interpretation of the program variable x in the first and second memories. Assertions in pRHL are deterministic and do not refer to probabilities.

Definition 3. *A* pRHL judgment *is a quadruple of the form* $\vDash c_1 \sim c_2 : \Psi \Rightarrow \Phi$*, where* Ψ *and* Φ *are assertions, and* c_1 *and* c_2 *are separable statements, i.e. they do not have any variable in common. A judgment is* valid *iff for every memories* m_1 *and* m_2*, we have* $(m_1, m_2) \models \Psi \Rightarrow ([\![c_1]\!](m_1), [\![c_2]\!](m_2)) \models \Phi^\dagger$.

Judgments can be proved valid with a variety of rules.

Two-sided and one-sided rules. The pRHL logic features two-sided rules (Figure 1) and one-sided rules (Figure 2); roughly speaking, two-sided rules relate two commands with the same structure and control flow, while one-sided rules relate two commands with possibly different structure or control flow; these rules allow pRHL to express *asynchronous* couplings between programs that may exhibit different control flow.

We point out two rules that will be especially important for our purposes. The rule [Sample] is used for relating two sampling commands. Note that it requires an injective function $f : T_1 \to T_2$ from the domain of the first sampling command to the domain of the second sampling command. When the two sampling commands have the same domain—as will be the case in our examples—f is simply a bijection on $T = T_1 = T_2$. This bijection gives us the freedom to specify the relation between the two samples.

The rule [While] is the standard while rule adapted to pRHL. Note that we require the guard of the two commands to be equal—so in particular the two loops must make the same number of iterations—and Φ plays the role of the while loop invariant as usual.

$$\textsc{SampleL}\ \frac{\vDash \mathsf{skip} \sim c : \forall v, \Psi[v/x_1] \Rightarrow \Phi}{\vDash x_1 \xleftarrow{\$} d_1 \sim c : \Psi \Rightarrow \Phi}$$

$$\textsc{IfL}\ \frac{\vDash c_1 \sim c : \Psi \wedge e \Rightarrow \Phi \qquad \vDash c_1' \sim c : \Psi \wedge \neg e_1 \Rightarrow \Phi}{\vDash \mathsf{if}\ e_1\ \mathsf{then}\ c_1\ \mathsf{else}\ c_1' \sim c : \Psi \Rightarrow \Phi}$$

$$\textsc{WhileL}\ \frac{\vDash c_1 \sim c : \Phi \wedge e_1 \Rightarrow \Phi \qquad \mathsf{while}\ e_1\ \mathsf{do}\ c_1\ \text{lossless}}{\vDash \mathsf{while}\ e_1\ \mathsf{do}\ c_1 \sim c : \Phi \Rightarrow \Phi}$$

Fig. 2: One-sided proof rules (selection)

Structural and program transformation rules. pRHL also features structural rules that are very similar to those of Hoare logic, including the rule of consequence and the case rule. In addition, it features a rule for program transformations, based on an equivalence relation $\simeq$ that provides a sound approximation of semantical equivalence. For our examples, it is sufficient that the relation $\simeq$ models loop range splitting and biased coin splitting, as given by the following clauses:

$$\mathsf{while}\ e\ \mathsf{do}\ c \simeq \mathsf{while}\ e \wedge e'\ \mathsf{do}\ c; \mathsf{while}\ e\ \mathsf{do}\ c$$
$$x \xleftarrow{\$} \mathbf{Bern}(p_1 \cdot p_2) \simeq x_1 \xleftarrow{\$} \mathbf{Bern}(p_1); x_2 \xleftarrow{\$} \mathbf{Bern}(p_2); x \leftarrow x_1 \wedge x_2$$

Figure 3 provides a selection of structural and program transformation rules.

$$\textsc{Conseq}\ \frac{\vDash c_1 \sim c_2 : \Psi' \Rightarrow \Phi' \qquad \Psi \Rightarrow \Psi' \qquad \Phi' \Rightarrow \Phi}{\vDash c_1 \sim c_2 : \Psi \Rightarrow \Phi}$$

$$\textsc{Case}\ \frac{\vDash c_1 \sim c_2 : \Psi \wedge \Psi' \Rightarrow \Phi \quad \vDash c_1 \sim c_2 : \Psi \wedge \neg\, \Psi' \Rightarrow \Phi}{\vDash c_1 \sim c_2 : \Psi \Rightarrow \Phi}$$

$$\textsc{Equiv}\ \frac{\vDash c_1' \sim c_2' : \Psi \Rightarrow \Phi \qquad c_1 \simeq c_1' \qquad c_2 \simeq c_2'}{\vDash c_1 \sim c_2 : \Psi \Rightarrow \Phi}$$

Fig. 3: Structural and program transformation rules (selection)

2.2 From pRHL judgments to probability judgments

We will derive two kinds of program properties from the existence of an appropriate probabilistic coupling. We will first discuss the mathematical theorems, where the notation is lighter and the core idea more apparent, and then demonstrate how the mathematical version can be expressed in terms of pRHL judgments.

Total variation and coupling. The first principle bounds the distance between two distributions in terms of a probabilistic coupling. We first define the total variation distance, also known as statistical distance, on distributions.

Definition 4. *Let X and X' be distributions over a countable set A. The* total variation *(TV) distance between X and X' is defined by* $\|X - X'\|_{tv} \triangleq \frac{1}{2} \sum_{a \in A} |X(a) - X'(a)|$.

To bound the distance between two distributions, it is enough to find a coupling and bound the probability that the two coupled variables differ.

Theorem 1 (Total variation, see [7]). *Let X and X' be distributions over a countable set. Then for any coupling $Y = (\hat{X}, \hat{X}')$, we have*

$$\|X - X'\|_{tv} \leq \Pr_{(x,x') \sim Y}[x \neq x'].$$

This theorem is useful for reasoning about convergence of distributions.

We can find a pRHL analog of this theorem. We first introduce some useful notation. For every expression e of type T and distribution μ over memories, let $[\![e]\!]_\mu$ be defined as $\mathsf{Mlet}\ m = \mu\ \mathsf{in}\ \mathsf{unit}\ m(e)$; note that $[\![e]\!]_\mu$ denotes a *distribution* over T. Similarly, for every event E (modeled as a predicate over memories) and distribution μ over memories, let $[\![E]\!]_\mu$ be defined as $\mathsf{Mlet}\ m = \mu\ \mathsf{in}\ \mathsf{unit}\ E(m)$. Thus, $[\![E]\!]_\mu$ is the probability of event E holding in the distribution μ.

Proposition 1. *If $\vDash c_1 \sim c_2 : \Psi \Rightarrow \Phi \Rightarrow v_1 = v_2$, where Φ exclusively refers to variables in c_1, then for every initial memories m_1 and m_2 that satisfy the precondition, the statistical distance between $[\![v_1]\!]_{[\![c]\!](m_1)}$ and $[\![v_2]\!]_{[\![c]\!](m_2)}$ is upper bounded by $\neg\Phi$ in $[\![\neg\Phi]\!]_{[\![c]\!](m_1)}$, i.e. $\|[\![v_1]\!]_{[\![c_1]\!](m_1)} - [\![v_2]\!]_{[\![c_2]\!](m_2)}\|_{tv} \leq [\![\neg\Phi]\!]_{[\![c]\!](m_1)}$.*

This proposition underlies the "up-to-bad" reasoning in EasyCrypt.

Stochastic domination and coupling. A second relational property of distributions is *stochastic domination*.

Definition 5. *Let X and X' be distributions over set A with an order relation $\geq$. We say X* stochastically dominates *X', written $X \geq_{sd} X'$, if for all $a \in A$,*

$$\Pr_{x \sim X}[x \geq a] \geq \Pr_{x' \sim X'}[x' \geq a].$$

Intuitively, stochastic domination defines a partial order on distributions over A given an order over A. Strassen's theorem shows that stochastic dominance is intimately related to coupling.

Theorem 2 (Strassen's theorem, see Lindvall [7]). *Let X and X' be distributions over a countable ordered set A. Then $X \geq_{sd} X'$ if and only if there is a coupling $Y = (\hat{X}, \hat{X}')$ with $Y \in \mathfrak{L}_{\geq}(X, X')$.*

We can express the "if" direction in the following pRHL form.

Proposition 2. *If $\vDash c_1 \sim c_2 : \Psi \Rightarrow v_1 \geq v_2$, then for every initial memories m_1 and m_2 that satisfy the precondition, $[\![v_1]\!]_{[\![c]\!](m_1)} \geq_{sd} [\![v_2]\!]_{[\![c]\!](m_2)}$.*

3 Warming up: Random walks

We warm up with couplings for random walks. These numeric processes models the evolution of a token over a discrete space: at each time step the token will choose its next movement randomly. We will show that starting from any two positions, the distributions of the two positions converges as we take more steps.

3.1 The basic random walk

Our first example is a random walk on the integers. Starting at an initial position, at each step we flip a fair coin. If heads, we move one step to the right. Otherwise, we move one step to the left. The code for running process k steps is presented in the left side of Figure 4. The variable `H` stores the history of coin flips. While this history isn't needed for computation of the result (it is *ghost code*), this history will be useful for stating invariants about the process.

```
pos ← start; H ← []; i ← 0;
while i < k do
  b ←$ {0,1};
  H ← b :: H;
  if b then pos++ else pos-- fi;
  i ← i + 1;
end
return pos
```

(a) Random walk on $\mathbb{Z}$

```
pos ← start; H ← []; i ← 0;
while i < k do
  mov ←$ {0,1};
  dir ←$ {0,1};
  crd ←$ [1,d];
  H ← (mov, dir, crd) :: H;
  if mov then
    pos ← pos + (dir ? 1 : -1 ) * u(crd)
  fi;
  i   ← i + 1;
end
return pos
```

(b) Random walk on $(\mathbb{Z}/k\mathbb{Z})^d$

Fig. 4: Two random walks

We consider two walks that start at locations $\mathtt{start}_1$ and $\mathtt{start}_2$, such that $\mathtt{start}_2 - \mathtt{start}_1 = 2n \geq 0$. We want to show that the distribution on end positions in the two walks converges as k increases. From Theorem 1, it suffices to find a coupling of the two walks, i.e., a way to coordinate their random samplings.

The basic idea is to *mirror* the two walks. When the first process moves towards the second process, we have the second process also move closer; when the first process moves away, we have the second process move away too. When the two processes meet, we have the two processes make identical moves.

To carry out this plan, we define $\Sigma(\mathtt{H})$ to be the number of `true` in `H` minus the number of `false`; in terms of the random walk, $\Sigma(\mathtt{H})$ measures the net change in position of a process with history `H`. Then, we define a predicate such that $P(\mathtt{H})$ holds when `H` contains a prefix `H'` with that $\Sigma(\mathtt{H}') = n$.

Accordingly, $P(\mathtt{H}_1)$ holds when the first process has moved at least n spots to the right. Under the coupling, this means that the second process must have

moved at least n spots to the left since the two particles are mirrored. Since the first process starts out exactly $2n$ to the left of the second process, $P(\mathsf{H}_1)$ is true exactly when the coupled processes have already met.

To formalize this coupling in pRHL, we aim to couple two copies of the program above, which we denote c_1 and c_2. We relate the two while loops with rule [While] using the following invariant:

$$(\mathsf{pos}_1 \neq \mathsf{pos}_2 \Rightarrow \mathsf{pos}_1 = \mathsf{i}_1 + \Sigma(\mathsf{H}_1) \wedge \mathsf{pos}_2 = \mathsf{i}_2 - \Sigma(\mathsf{H}_1)) \wedge (P(\mathsf{H}_1) \Rightarrow \mathsf{pos}_1 = \mathsf{pos}_2).$$

The loop invariant states that before the two particles meet, their trajectories are mirrored, and that once they have met, they coincide forever.

To prove that this is an invariant, we need to relate the loop bodies. The key step is relating the two sampling operations using the rule [Sample]; note that we must provide a bijection f from booleans to booleans. We choose the bijection based on whether the two coupled walks have met or not.

More precisely, we perform a case analysis on $\mathsf{pos}_1 = \mathsf{pos}_2$ with rule [Case]. If they are equal then the walks move together, so we use the identity map for f; this has the effect of forcing both processes to see the same sample. If the walks are at different positions, we use the negation map $(\neg)$ for f, so as to force the two processes to take opposite steps.

Putting everything together, we can prove the following judgment in pRHL:

$$\vDash c_1 \sim c_2 : \mathsf{start}_1 + 2n = \mathsf{start}_2 \Rightarrow (P(\mathsf{H}_1) \Rightarrow \mathsf{pos}_1 = \mathsf{pos}_2).$$

By Theorem 1, we can bound the TV distance between the final positions. If two memories m_1, m_2 satisfy $m_1(\mathsf{start}) + 2n = m_2(\mathsf{start})$, we have

$$\|[\![\mathsf{pos}_1]\!]_{[\![c_1]\!](m_1)} - [\![\mathsf{pos}_2]\!]_{[\![c_2]\!](m_2)}\|_{tv} \leq [\![\neg P(\mathsf{H}_1)]\!]_{[\![c_1]\!](m_1)}.$$

Note that the right hand side depends only on the first program. In other words, proving this quantitative bound on two programs is reduced to proving a quantitative property on a *single* program—this is the power of coupling.

3.2 Lazy random walk on a torus

For a more interesting example of a random walk, we can consider a walk on a torus. Concretely, the position is now a d-tuple of integers in $[0, k-1]$. The walk first flips a fair coin; if heads it stays put, otherwise it moves. If it moves, the walk chooses uniformly in $[1, d]$ to choose the coordinate to move, and a second fair coin to determine the direction (positive, or negative). The positions are cyclic: increasing from $k-1$ leads to 0, and decreasing from 0 leads to $k-1$.

We can simulate this walk with the program in the right side of Figure 4, where $\mathsf{u}(i)$ is the i-th canonical base vector in $(\mathbb{Z}/k\mathbb{Z})^d$. As before, we store the trace of the random walk in the list H. All arithmetic is done modulo k.

Like the simple random walk, we start this process at two locations start_1 and start_2 on the torus and run for k iterations. We aim to prove that the distributions of the two walks converge as k increases by coupling the two walks, iteration by

iteration. Each iteration, we first choose the same coordinate `crd` and the same direction `dir` in both walks. If the two positions coincide in coordinate `crd`, we arrange both walks to select the same direction `mov`, so that the walks either move together, or both stay put. If the two positions differ in `crd`, we arrange the walks to select opposite samples in `mov` so that exactly one walk moves.

As in the basic random walk, we can view our coupling as letting the first process evolve as usual, then coordinating the samples of the second process to perform the coupling. In other words, given a history $\mathtt{H}_1$ of samples for the first process, the behavior of the second coupled process is completely specified.

Thus, we can define operators to extract the movements of each walk from the trace $\mathtt{H}_1$ of the samplings of the first process: $\Sigma_1(i, \mathtt{H}_1)$ is the drift of the ith coordinate of the first process, and $\Sigma_2(i, \mathtt{H}_1)$ is the drift of the second process. Essentially, these operators encode the coupling by describing how the second process moves as a function of the first process's samples.

In pRHL, we will use the rule [While] with the following invariant:

$$\forall i \in [1, d].\ (\Sigma_1(i, \mathtt{H}_1) - \Sigma_2(i, \mathtt{H}_1) = \Delta[i] \Rightarrow \mathtt{pos}_1[i] = \mathtt{pos}_2[i])$$
$$\wedge\ (\mathtt{pos}_1[i] \neq \mathtt{pos}_2[i] \Rightarrow \mathtt{pos}_1[i] = \mathtt{start}_1[i] + \Sigma_1(i, \mathtt{H}_1) \wedge \mathtt{pos}_2[i] = \mathtt{start}_2[i] + \Sigma_2(i, \mathtt{H}_1)),$$

where Δ is the vector $\mathtt{start}_2 - \mathtt{start}_1$. The invariant describes the position of the two coupled processes in terms of the history $\mathtt{H}_1$. We are particularly interested in the first conjunct, which gives a condition for the two coupled processes to meet in coordinate i.

To prove that the invariant is preserved, we encode the coupling described above into pRHL, via three uses of the rule [Sample]. The first two samples—for `crd` and `dir`—are coupled with f being identity bijections (on $[1, d]$ and on booleans), ensuring that the processes make identical choices. When sampling `mov`, we inspect the history $\mathtt{H}_1$ to see whether the two walks agree in position `crd`. If so, we choose the identity bijection for `mov`; if not, we choose negation. This coupling is sufficient to verify the loop invariant.

To conclude our proof, the first conjunct in the invariant implies that we can prove the pRHL judgment $\vDash c_1 \sim c_2 : \mathtt{start}_2 - \mathtt{start}_1 = \Delta \Rightarrow \Phi$, where

$$\Phi \triangleq (\forall i \in [1, d].\ \Sigma_1(i, \mathtt{H}_1) - \Sigma_2(i, \mathtt{H}_1) = \Delta[i]) \Rightarrow \forall i \in [1, d].\ \mathtt{pos}_1[i] = \mathtt{pos}_2[i].$$

Finally, Theorem 1 implies that for any two initial memories m_1, m_2 with $m_2(\mathtt{start}) - m_1(\mathtt{start}) = \Delta$, we have

$$\|[\![\mathtt{pos}_1]\!]_{[\![c_1]\!](m_1)} - [\![\mathtt{pos}_2]\!]_{[\![c_2]\!](m_2)}\|_{tv} \leq [\![\exists i \in [1, d].\ \Sigma_1(i, \mathtt{H}_1) - \Sigma_2(i, \mathtt{H}_1) \neq \Delta[i]]\!]_{[\![c_1]\!](m_1)}.$$

Again, proving a quantitative bound on the convergence of two distributions is reduced to proving a quantitative bound on a single program.

4 Combining coupling with program transformation

So far, we have seen examples where the coupling is proved directly on the two original programs c_1 and c_2. Often, it is convenient to introduce a third program c^* that is equivalent to c_1, and then couple c^* to c_2. Applying transitivity (rule [Equiv]), this gives a coupling between c_1 and c_2. Let's consider two examples.

4.1 Two biased coins

Consider a coin flipping process that flips a coin k times, and returns the number of heads observed. We consider this process run on two different biased coins: The first coin has probability q_1 of coming up heads, while the second coin has probability q_2 of coming up heads with $q_1 \geq q_2$. Let the distribution on the number of heads be μ_1 and μ_2 respectively.

Intuitively, it is clear that the first process is somehow bigger than the second process: it is more likely to see more heads, since the first coin is biased with a higher probability. Stochastic dominance turns out to be the proper way to formalize our intuition. To prove it, Proposition 2 implies that we just need to find an appropriate coupling of the two processes.

While it is possible to define a coupling directly by carefully coordinating the corresponding coin flips, we will give a simpler coupling that proceeds in two stages. First, we will couple a program c_1 computing μ_1 to an intermediate program c^*. Then, we will show that c^* is equivalent to a program c_2 computing μ_2, thus exhibiting a coupling between μ_1 and μ_2. Letting $r = q_2/q_1$ and denoting the coin flip distribution with probability p of sampling true by $\mathbf{Bern}(p)$, we give the programs in Figure 5.

For the first step, we want to couple c_1 and c^*. For a rough sketch, we want to use rule [While] with an appropriate loop invariant; here, $n_1 \geq n^*$. To show that the invariant is preserved, we need to relate the loop bodies. We use the two-sided rule [Sample] when sampling x and y (taking the bijection f to be the identity), the one-sided rule [Sample-L] to relate sampling nothing (skip) in c_1 with sampling z in c^*, and the one-sided rule [IfL] to relate the two conditionals. (The one-sided rule is needed, since the two conditionals may take different branches.) Thus, we can prove the judgment $\vDash c_1 \sim c^* : q_1 \geq q_2 \wedge r = q_2/q_1 \Rightarrow n_1 \geq n^*$.

For the second step, we need to prove that c^* is equivalent to c_2. Here, we use a sound approximation $\simeq$ to semantic equivalence as described in the introduction. Specifically, we have $\mathsf{x} \xleftarrow{\$} \mathbf{Bern}(q_1 \cdot r) \simeq \mathsf{y} \xleftarrow{\$} \mathbf{Bern}(q_1); \mathsf{r} \xleftarrow{\$} \mathbf{Bern}(r); \mathsf{x} \leftarrow \mathsf{y} \wedge \mathsf{z}$ for the loop bodies; showing equivalence of c^* and c_2 is then straightforward. Thus, we can show $\vDash c^* \sim c_2 : q_1 \geq q_2 \wedge r = q_2/q_1 \Rightarrow n^* = n_2$. Applying rule [Equiv] gives the final judgment $\vDash c_1 \sim c_2 : q_1 \geq q_2 \wedge r = q_2/q_1 \Rightarrow n_1 \geq n_2$, showing stochastic domination by Proposition 2.

4.2 Balls into bins: asynchronous coupling

The examples we have seen so far are all *synchronous* couplings: they relate the iterations of the while loop in lock-step. For some applications, we may want to reason asynchronously, perhaps allowing one side to progress while holding the other side fixed. One example of an asynchronous coupling is analyzing the *balls into bins* process. We have two bins, and a set of n balls. At each step, we throw a ball into a random bin, returning the count of both bins when we have thrown all the balls. The code is on the left side in Figure 6.

Now, we would like to consider what happens when we run two processes with different numbers of balls. Intuitively, it is clear that if the first process throws

```
n ← 0; i ← 0;
whil i < k do:
  x ←$ Bern(q1);
  if x then
    n ← n + 1;
  fi
  i ← i + 1;
end
return n
```

(a) Program c_1

```
n ← 0; i ← 0;
while i < k do:
  y ←$ Bern(q1);
  z ←$ Bern(r);
  x ←$ y ∧ z;
  if x then
    n ← n + 1;
  fi;
  i ← i + 1;
end
return n
```

(b) Program c^*

```
n ← 0; i ← 0;
while i < k do:
  x ←$ Bern(q2);
  if x then
    n ← n + 1
  fi;
  i ← i + 1
end
return n
```

(c) Program c_2

Fig. 5: Coupling for biased coin flips

```
i, binA, binB ← 0;
while i < n do
  i ← i + 1;
  b ←$ {0,1};
  if b then binA++ else binB++ fi
end
return (binA, binB)
```

(a) Original programs c_1, c_2

```
i, binA, binB ← 0;
while i < n ∧ i < m do
  b ←$ {0,1};
  if b then binA++ else binB++ fi;
  i ← i + 1;
end
while i < n do
  b ←$ {0,1};
  if b then binA++ else binB++ fi;
  i ← i + 1;
end
return (binA, binB)
```

(b) Intermediate program c^*

Fig. 6: Coupling balls into bins

more balls than the second process, it should result in a higher load in the bins; we aim to prove that the first process stochastically dominates the second with the following coupling. Assume that the first process has more balls ($n_1 \geq n_2$). For the first n_2 balls, we have the two process do the same thing—they choose the same bucket for their tosses. For the last $n_1 - n_2$ steps, the first process throws the rest of the balls. Evidently, this coupling forces the bins in the first run to have higher load than the bins in the second run.

To formalize this example, we again introduce a program c^*, proving equivalence with c_1 and showing a coupling with c_2. The code for c^* is on the right side in Figure 6; we require the dummy input `m` to be equal to $\mathtt{n}_2$.

Proving equivalence with program c_1 is direct, using the loop range splitting transformation in EasyCrypt: $\mathsf{while}\ e\ \mathsf{do}\ c \simeq \mathsf{while}\ e \wedge e'\ \mathsf{do}\ c; \mathsf{while}\ e\ \mathsf{do}\ c$. Once this is done, we simply need to provide a coupling between c^* and c_2. By our choice of `m`, we can trivially couple the first loop in c^* to the (single) loop in c_2, ensuring that $\Phi \triangleq \mathtt{binA}^* \geq \mathtt{binA}_2 \wedge \mathtt{binB}^* \geq \mathtt{binB}_2$ after the first loop.

Then, we can apply the one-sided rules to couple the second loop in c^* with a skip statement in c_2. It is straightforward to show that Φ is an invariant in

rule [WhileL], from which we can conclude $\vDash c^* \sim c_2 : n_1 \geq n_2 \wedge m = n_2 \Rightarrow \texttt{binA}^* \geq \texttt{binA}_2 \wedge \texttt{binB}^* \geq \texttt{binB}_2$, and by equivalence of c_1 and c^* we have $\vDash c_1 \sim c_2 : n_1 \geq n_2 \Rightarrow \texttt{binA}_1 \geq \texttt{binA}_2 \wedge \texttt{binB}_1 \geq \texttt{binB}_2$, enough for stochastic domination by Proposition 2.

5 Non-deterministic couplings: birth and death

So far, we have seen *deterministic* couplings, which reuse randomness from the coupled processes in the coupling; this can be seen in the [Sample] rule, when we always choose a deterministic. In this section, we will see a more sophisticated coupling that injects new randomness.

For our example, we consider a classic Markov process. Roughly speaking, a Markov process moves within a set of states each transition depending only on the current state and a fresh random sample. The random walks we saw before are classic examples of Markov processes.

A more complex Markov process is the *birth and death chain*. The state space is $\mathbb{Z}$, and the process starts at some integer x. At every time step, if the process is at state i, the process has some probability b_i of increasing by one, and some probability a_i of decreasing by one. Note that a_i and b_i may add up to less than 1: there can be some positive probability $1 - a_i - b_i$ where the process stays fixed.

To model this process, we define a sum type `Move` with three elements (`Left`, `Right` and `Still`) which correspond to the possible moves a process can make. Then, the chains are modeled by the code in the left of Figure 7, where the distribution `bd(state)` is the distribution of moves from `state`.

```
H ← []; state ← start; i ← 0;
while i < k do
  dir ←$ bd(state);

  if dir = Left then
    state ← state - 1;
  else if dir = Right then
    state ← state + 1;
  fi
  H ← state :: H;
  i ← i + 1;
end
return state
```

(a) Original programs c_1, c_2

```
H ← []; state ← start; i ← 0;
while i < steps do
  d   ←$ dcouple;
  dir ← proj [1|2] d;
  if dir = Left then
    state ← state - 1;
  else if dir = Right then
    state ← state + 1;
  fi
  H ← state :: H;
  i ← i + 1;
end
return state
```

(b) Intermediate programs c_1^*, c_2^*

Fig. 7: Coupling the birth and death chain

Just like the biased coin and balls into bins processes, we want to prove stochastic domination for two processes started at states $\texttt{start}_1 \geq \texttt{start}_2$ via coupling. The difficulty is that if the processes become adjacent and they both move, the two processes may swap positions, losing stochastic domination.

The solution is to use a special coupling when the two processes are on two adjacent states as in Mufa [8]. Unlike the previous examples, the coupling is not deterministic: the behavior of one process is not fully determined by the randomness of the other. Our loop invariant is the usual one for stochastic domination: $\texttt{state}_1 \geq \texttt{state}_2$. To show that this invariant is preserved, we perform a case analysis on whether $\texttt{state}_1 = \texttt{state}_2$, $\texttt{state}_1 = \texttt{state}_2 + 1$ or $\texttt{state}_1 > \texttt{state}_2 + 1$.

We focus on interesting case: the middle one, when the states are adjacent. Here, we perform a trick: we switch c_1, c_2 for two equivalent intermediate programs c_1^*, c_2^*, and prove a coupling on the two intermediate programs. The two intermediate programs each sample from `dcouple`, a distribution on pairs of moves, and project out the first or second component as `dir`; in other words, we explicitly code c_1^*, c_2^* as sampling from the two marginals of a common distribution `dcouple`. By proving that the marginals are indeed distributed as $\texttt{bd}(\texttt{state}_1)$ and $\texttt{bd}(\texttt{state}_2)$, we can prove equivalences $c_1 \simeq c_1^*$ and $c_2 \simeq c_2^*$. The code is in the right side of Figure 7, where `proj [1|2]` is the first pair projection π_1 in c_1, and the second pair projection π_2 in c_2.

All that remains is to prove a coupling between c_1^* and c_2^* satisfying the loop invariant $\texttt{state}_1 \geq \texttt{state}_2$. With adjacent states, `dcouple` is given by the following function from pairs of moves to probabilities:

```
op distr-adjacent a_i  a_{i+1}  b_i  b_{i+1}  (x : Move * Move) =
  if x = (Right, Left ) then min(b_{i+1}, a_i)    else
  if x = (Still, Left ) then (b_{i+1} - a_i)^+    else
  if x = (Right, Still) then (a_i - b_{i+1})^+    else
  if x = (Still, Right) then a_{i+1}              else
  if x = (Left , Still) then b_i                  else
  if x = (Still, Still) then
    1 - min(b_{i+1}, a_i) - a_{i+1} - b_i - |b_{i+1} - a_i| else
  if x = (_    , _    ) then 0.
```

Note that the case `(Left, Right)` has probability 0: this forbids the first process from skipping past the second process.

Now the coupling is easy: we simply require both samples from `dcouple` to be the same. Since $\texttt{state}_1 = \texttt{state}_2 + 1$ and the distribution never returns $(\texttt{Left}, \texttt{Right})$, the loop invariant is trivially preserved. This shows the desired coupling, and stochastic domination by Proposition 2.

6 Conclusion and future work

We have established the connection between relational verification of probabilistic programs using pRHL, and coupling, which plays a major role in probability theory. Furthermore, we have used the connection by verifying in pRHL several well-known examples of couplings from the literature on randomized algorithms.

More broadly, our work occupies a middle ground between the two main approaches to relational verification: (i) encoding the two programs as a single program and reasoning about the single program (techniques include include cross-products [12], self-composition [1], and product programs [3]); and (ii) using a program logic to reason directly about two programs (techniques include relational Hoare logic [5], relational separation logic [11], and pRHL [2]).

Our work also suggests the possibility of a practical proof technique for verifying probability results based on coupling, but a deeper understanding of the links between pRHL and coupling (and their variants thereof), and a more general framework for verification of probabilistic programs will be required.

A more general verification framework. pRHL derivations implicitly construct a sequence of couplings between the executions—modeled as sequences of distributions—of the two programs involved in the final statement. The single most important step in the construction of the coupling is performed when applying the rule [Sample] for random sampling, and more specifically when picking the bijection function f. A careful look at the rule reveals that the coupling is a *deterministic* coupling, as defined by Villani [10]:

Definition 6. *A coupling (X, Y) between two spaces $\mathcal{X}, \mathcal{Y}$ is said to be* deterministic *if there exists a function $T : \mathcal{X} \to \mathcal{Y}$ such that $Y = T(X)$.*

Indeed, deterministic couplings are often sufficient in cryptographic proofs—the original motivation to develop pRHL. However, there are many examples of couplings that cannot be verified using deterministic couplings. For our examples we have worked around the difficulty by using program transformation rules. However, this approach is very specific, and does not generalize. Thus, an interesting generalization of pRHL would be to relax the requirement that the relationship between the two samplings is given by a 1-1 map, by allowing a binary relation which satisfies suitable conditions. Such a general rule could enable a more general class of couplings, and yield more principled proofs of some of the couplings studied in this paper.

Moreover, it would be interesting to extend EasyCrypt with mechanisms for handling the non-relational reasoning in couplings. To prove quantitative bounds on total variation in the random walk example, we need to bound the time it takes for a single random walk to reach a certain position. Obtaining precise bounds on the speed of convergence requires more complex reasoning, and will require an expressive program logic. Such a program logic is under development, but has not yet been integrated in EasyCrypt.

Extending to shift and path coupling. The couplings realized in the random walks are instances of exact couplings, i.e. the paths will eventually merge. A more general notion of coupling for such processes is the *shift-coupling*, where the merging condition is "relaxed" in the sense that the paths will merge modulo a random time shift. The general theory of path couplings provides similar-shaped inequalities as the ones in exact coupling, allowing powerful mathematical-based reasoning inside the logic with the [Conseq] rule. These coupling notions are complex, and it is not yet clear how they can be verified.

Other examples. The coupling literature is rich with further examples, from simple to challenging. We intend to verify many other examples of couplings, in particular the proof of Dynkin's card trick, and the proof of the constructive Lovasz Local Lemma, a fundamental tool used in the probabilistic method.

Bibliography

[1] G. Barthe, P. D'Argenio, and T. Rezk. Secure Information Flow by Self-Composition. In R. Foccardi, editor, *Computer Security Foundations, CSF'04*, pages 100–114. IEEE Press, 2004.

[2] G. Barthe, B. Grégoire, and S. Zanella Béguelin. Formal certification of code-based cryptographic proofs. In Z. Shao and B. C. Pierce, editors, *Principles of Programming Languages, POPL'09*, pages 90–101. ACM Press, 2009.

[3] G. Barthe, J. M. Crespo, and C. Kunz. Relational verification using product programs. In *Formal Methods*, Lecture Notes in Computer Science. Springer, 2011.

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

[5] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In N. D. Jones and X. Leroy, editors, *Principles of Programming Languages, POPL'04*, pages 14–25. ACM Press, 2004.

[6] Y. Deng and W. Du. Logical, metric, and algorithmic characterisations of probabilistic bisimulation. Technical Report CMU-CS-11-110, Carnegie Mellon University, March 2011.

[7] T. Lindvall. *Lectures on the coupling method*. Courier Corporation, 2002.

[8] C. Mufa. Optimal markovian couplings and applications. *Acta Mathematica Sinica*, 10(3):260–275, 1994.

[9] H. Thorisson. *Coupling, Stationarity, and Regeneration*. Springer, 2000.

[10] C. Villani. *Optimal transport: old and new*. Springer Science, 2008.

[11] H. Yang. Relational separation logic. *Theoretical Comput. Sci.*, 375(1-3): 308–334, 2007.

[12] A. Zaks and A. Pnueli. Covac: Compiler validation by program analysis of the cross-product. In *Formal Methods*, pages 35–51, 2008.

A Graphical depictions of random processes

We depict the two random walk processes in Figures 8 and 9.

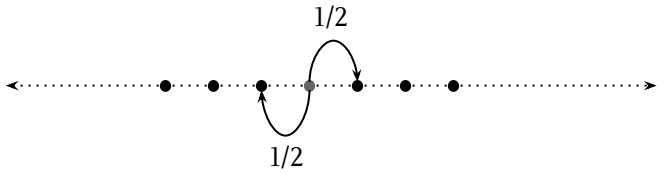

Fig. 8: Unbiased random walk

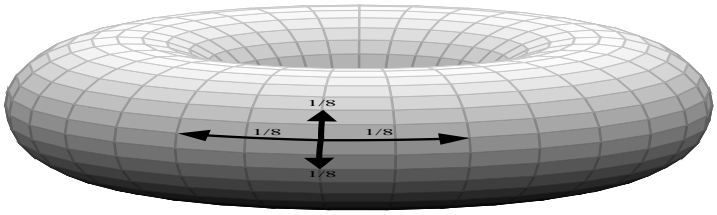

Fig. 9: Lazy random walk on a two dimensional torus