Approximate Relational Reasoning for Higher-Order Probabilistic Programs

PHILIPP G. HASELWARTER, Aarhus University, Denmark
KWING HEI LI, Aarhus University, Denmark
ALEJANDRO AGUIRRE, Aarhus University, Denmark
SIMON ODDERSHEDE GREGERSEN, New York University, USA
JOSEPH TASSAROTTI, New York University, USA
LARS BIRKEDAL, Aarhus University, Denmark

Properties such as provable security and correctness for randomized programs are naturally expressed relationally as approximate equivalences. As a result, a number of relational program logics have been developed to reason about such approximate equivalences of probabilistic programs. However, existing approximate relational logics are mostly restricted to first-order programs without general state.

In this paper we develop Approxis, a *higher-order approximate relational separation logic* for reasoning about approximate equivalence of programs written in an expressive ML-like language with discrete probabilistic sampling, higher-order functions, and higher-order state. The Approxis logic recasts the concept of *error credits* in the relational setting to reason about relational approximation, which allows for expressive notions of modularity and composition, a range of new approximate relational rules, and an internalization of a standard limiting argument for showing exact probabilistic equivalences by approximation. We also use Approxis to develop a logical relation model that quantifies over error credits, which can be used to prove *exact contextual equivalence*. We demonstrate the flexibility of our approach on a range of examples, including the PRP/PRF switching lemma, IND$-CPA security of an encryption scheme, and a collection of rejection samplers. All of the results have been mechanized in the Coq proof assistant and the Iris separation logic framework.

CCS Concepts: • **Theory of computation** → **Separation logic**; **Logic and verification**; **Probabilistic computation**; **Program verification**; • **Mathematics of computing** → **Probabilistic algorithms**.

Additional Key Words and Phrases: Probabilistic Couplings, Separation Logic, Logical Relations

ACM Reference Format:
Philipp G. Haselwarter, Kwing Hei Li, Alejandro Aguirre, Simon Oddershede Gregersen, Joseph Tassarotti, and Lars Birkedal. 2025. Approximate Relational Reasoning for Higher-Order Probabilistic Programs. *Proc. ACM Program. Lang.* 9, POPL, Article 41 (January 2025), 35 pages. https://doi.org/10.1145/3704877

1 Introduction

Many important properties of probabilistic programs are naturally expressed as *approximate* equivalence of two programs. For example, provable security [Goldwasser and Micali 1984] compares an implementation of a cryptographic scheme to an idealized specification program that does not have access to any sensitive information, and aims to show that an adversary can only distinguish them with some small probability. In a similar spirit, many randomized algorithms and data structures

Authors' Contact Information: Philipp G. Haselwarter, Aarhus University, Denmark, pgh@cs.au.dk; Kwing Hei Li, Aarhus University, Denmark, hei.li@cs.au.dk; Alejandro Aguirre, Aarhus University, Denmark, alejandro@cs.au.dk; Simon Oddershede Gregersen, New York University, USA, s.gregersen@nyu.edu; Joseph Tassarotti, New York University, USA, jt4767@cs.nyu.edu; Lars Birkedal, Aarhus University, Denmark, birkedal@cs.au.dk.

ACM 2475-1421/2025/1-ART41
https://doi.org/10.1145/3704877

can be specified by showing that they are approximately equivalent to their non-probabilistic counterparts. Consequently, it is important to be able to reason about approximate equivalences and so a number of relational program logics have been developed for first-order languages [Barthe et al. 2017, 2016c, 2012] or higher-order languages with first-order global state [Aguirre et al. 2021].

In this work, we develop Approxis, a *higher-order approximate relational separation logic* for reasoning about approximate equivalence of RandML programs, an expressive ML-like language with discrete random sampling, higher-order functions, and higher-order dynamically-allocated state. A key point is that Approxis, inspired by the unary Eris logic [Aguirre et al. 2024], introduces *error credits* in the relational setting to reason about approximation. Error credits are separation-logic resources that bound the maximum approximation error between two programs. We introduce a collection of novel *approximate coupling rules*, which consume error credits in order to relate randomized transitions of two programs. By treating the relational approximation error as just another separation-logic resource, Approxis provides modular reasoning principles that enable more precise error accounting when composing proofs, much as Eris demonstrated in the non-relational setting.

Surprisingly, error credits not only allow us to prove approximate equivalences, they also allow us to prove *exact* equivalences that were beyond the scope of prior coupling-based relational program logics. Just as in real analysis, where one can prove two numbers are equal by showing that the distance between them is smaller than ε for all $\varepsilon > 0$, we can similarly show two probability distributions are equivalent by showing the distance between them is bounded by ε for all $\varepsilon > 0$. Using Approxis, we show how to recover this technique internally in the logic through *error amplification* [Aguirre et al. 2024] and thus prove exact equivalence of probabilistic programs by means of approximation. Based on this, we develop a new binary logical relations model of a rich type system for RandML with recursive types and impredicative polymorphism. The model supports approximate reasoning and gives us a powerful and novel method for showing exact *contextual* equivalence of higher-order probabilistic programs. For other existing approaches, including both operational approaches, *e.g.*, Clutch [Gregersen et al. 2024], and denotational approaches, *e.g.*, pRHL [Barthe et al. 2009] and HO-RHL [Aguirre et al. 2021], some of the examples that we consider would be very complicated—if not impossible—to handle.

We show that Approxis scales to more involved approximate reasoning by showing the classical PRP/PRF Switching Lemma [Bellare and Rogaway 2004; Hall et al. 1998] and IND\$-CPA security of a PRF-based symmetric encryption scheme. Moreover, we apply error amplification and our logical relation to show contextual equivalences for a collection of rejection samplers, including a sampling scheme for drawing a random sample from a B+ tree [Bayer and McCreight 1972].

Examples like the PRP/PRF Switching Lemma have been verified in many different settings, but we emphasize the rich programming language we consider here. While some of these examples might be expressible in simpler languages, features such as higher-order functions, higher-order state, and polymorphism are all found in general-purpose programming languages, and are needed for modern compositional software development. Moreover, cryptographic security can be more naturally expressed in such higher-order languages and avoids the need for syntactic restrictions on adversaries as seen, *e.g.*, in EasyCrypt [Barthe et al. 2014]. As a consequence, verification frameworks must handle these language features to reason about large applications and realistic implementations. Higher-order separation logic is a powerful and well-tested abstraction for this purpose, and Approxis shows how to beneficially apply it for approximate relational reasoning. While the B+ tree case study, for example, is quite involved, the complexity is managed through mostly-standard separation-logic reasoning. We see this as a significant strength of our approach.

At a technical level, our development builds upon the (non-approximate) probabilistic coupling logic Clutch [Gregersen et al. 2024]. By incorporating error credits [Aguirre et al. 2024] in the relational setting, our development generalizes the approach to approximate reasoning using approximate couplings. In addition, we introduce two new *coupling precondition* connectives and a notion of *erasability.* The erasability condition not only captures the soundness of asynchronous couplings [Gregersen et al. 2024] in a more semantic way, but also allows for a more principled approach to validating the new approximate and non-approximate coupling rules we introduce and which are critical for the examples that we consider.

Contributions. In summary, we make the following contributions:

- The first higher-order approximate relational separation logic, Approxis, for reasoning about approximate equivalence of RandML programs, an expressive ML-like language with probabilistic sampling, higher-order functions, and higher-order state,
- A logical internalization of a limiting argument that allows us to show exact equivalence of higher-order probabilistic programs through approximation,
- A class of new approximate and non-approximate coupling rules, including the *many-to-one* and *fragmented* coupling rules,
- A logical relations model of an expressive type system for RandML with recursive types and impredicative polymorphism, which allows us to show (exact) *contextual* equivalence of probabilistic programs through a limiting argument,
- A collection of case studies: the PRP/PRF Switching Lemma [Bellare and Rogaway 2004; Hall et al. 1998], IND$-CPA security of a PRF-based symmetric encryption scheme, and contextual equivalence of a selection of rejection samplers, including a sampling scheme for drawing a random sample from a B+ tree [Bayer and McCreight 1972]. Several of these are, to the best of our knowledge, beyond the scope of previous techniques, in particular for expressive languages such as RandML.
- Full mechanization of all results in the Coq proof assistant [Team 2024], building on top of the Iris separation logic framework [Jung et al. 2018] and the Coquelicot [Boldo et al. 2015] library for real analysis.

Outline. In §2 we give high-level intuition for how to reason using Approxis. Here we discuss the PRP/PRF Switching Lemma, a classical result in cryptography, and show how to use the limiting argument on a simple rejection sampler. In §3 we recall some definitions from probability theory and define the semantics of RandML. In §4 present a collection of program logic rules and coupling rules of Approxis before developing our logical relations model in §5. In §6 we showcase Approxis on a range of case studies, and in §7 we explain how the semantic model of Approxis is constructed on top of the Iris base logic. Finally, we discuss related work and conclude in §8 and §9, respectively.

2 Key Ideas

In this section, we give a high-level overview of Approxis and introduce how error credits can be used to do approximate relational reasoning. The primary specification assertion in Approxis is the *refinement weakest precondition*, written $\mathsf{rwp}\ e_1 \precsim e_2\ \{\Phi\}$, where e_1 and e_2 are two randomized programs, and Φ is a relation on the return values and final program states of e_1 and e_2. Informally, this relational connective says that if executing e_1 terminates with a value v_1, then e_2 terminates with value v_2 and the postcondition $\Phi(v_1, v_2)$ holds.

Because Approxis is a separation logic, when presenting the rules of the logic, we use inference rule-style notation with premises $P_1, \dots, P_n$ and conclusion Q to stand for the entailment $P_1 * \dots * P_n \vdash Q$ in the logic.

For reasoning about non-randomized steps of e_1 and e_2, Approxis has a variety of rules that are relational generalizations of usual separation logic rules, as in prior relational Hoare logics [Benton

2004; Frumin et al. 2021; Turon et al. 2013]. For randomized steps, the first tool Approxis provides are the so-called *coupling* rules pioneered by pRHL [Barthe et al. 2009]. A simple, specialized form of such a rule is

$$\frac{\forall n \leq N.\, \mathsf{rwp}\ n \precsim n\ \{\Phi\}}{\mathsf{rwp}\ \ \mathsf{rand}\, N \precsim \mathsf{rand}\, N\ \{\Phi\}}\ \textsc{wp-couple-exact}$$

where $\mathsf{rand}\, N$ is a command in the language that samples a value uniformly from $\{0, \ldots, N\}$. This rule says that if both programs are sampling from $\mathsf{rand}\, N$, then we may reason *as if* they both returned the same sample value n, instead of having to consider all $(N+1)^2$ possible combinations of values they could have returned. This rule is justified by using the notion of couplings from probability theory, and relies on the fact that the two sets being sampled from have the same size.

What if we want to reason about the case where the two sets being sampled from are *not* the same size? For example, suppose the left program executes $\mathsf{rand}\, N$ and the right executes $\mathsf{rand}\ (N+1)$. We cannot exactly reason as if both programs sample the same value: there is a chance that the program on the right samples $N+1$, which the program on the left can never do! However, the right program only draws this "bad" value of $N+1$ with probability $1/(N+2)$. If N is very large, this probability will be small, so we might hope to argue that we can approximately reason as if the two samples returned the same value, recovering an analogue of wp-couple-exact.

This idea of approximate relational reasoning has been developed in apRHL [Barthe et al. 2012]. In apRHL, relational Hoare triples are annotated with an additional parameter, ε, which bounds the approximation error.[1] Then, the coupling rules allow for relating two sampling commands from distributions that are only equal up to some error ε' by adding ε' to the total error on the Hoare triple. However, Aguirre et al. [2024] have previously shown that tracking an error bound as an additional parameter of a Hoare judgement has a number of limitations related to modularity and precision of bounds. Instead, they proposed to track errors through a separation logic assertion called an *error credit*, written $\lightning(\varepsilon)$, which represents a "permission" to incur an approximation error of up to ε. They developed this idea in a *unary* logic called Eris for bounding the probabilities of events of a randomized program. A key aspect of the flexibility of error credits arises from the fact that they can be split and joined, in the sense that $\lightning(\varepsilon_1 + \varepsilon_2) \dashv\vdash \lightning(\varepsilon_1) * \lightning(\varepsilon_2)$ for $\varepsilon_1, \varepsilon_2 \geq 0$.

Approxis uses this idea of error credits to track approximation error in couplings. A special case of Approxis's approximate coupling rule applied to the scenario described above would be:

$$\frac{\lightning\left(\frac{1}{N+2}\right) \qquad \forall n \leq N.\, \mathsf{rwp}\ n \precsim n\ \{\Phi\}}{\mathsf{rwp}\ \ \mathsf{rand}\, N \precsim \mathsf{rand}\ (N+1)\ \{\Phi\}}$$

which says that if we spend $\lightning\left(\frac{1}{N+2}\right)$ credits we may reason as if the two samples returned the same value. Informally, we think of the error credits as being spent to "rule out" the case where the program on the right returns $N+1$.

In Approxis a derivation of the form $\lightning(\varepsilon) \vdash \mathsf{rwp}\ e_1 \precsim e_2\ \{\Phi\}$ implies that at most ε total error is incurred in deriving the refinement weakest precondition. The soundness theorem for the logic then says that to prove that the distributions corresponding to two programs are within ε distance of one another (in a sense to be made precise later), it suffices to prove the refinement weakest preconditions in both directions, each with error up to ε.

At a high level, tracking approximate coupling error using credits seems like a relatively simple adaptation of Eris [Aguirre et al. 2024] to the relational setting. However, as we shall see later, doing so in a sound manner involves addressing several new technical challenges that have no analogue in the unary case. But first we shall look at an example of how the features of Approxis can be used to reason about cryptographic security.

[1] apRHL has a second annotation for bounding another form of probabilistic approximation which we do not consider.

2.1 Motivating Example: PRP/PRF Switching Lemma

To illustrate the different ideas coming together in Approxis, we explore a classic approximate equivalence result from cryptography: the PRP/PRF Switching Lemma [Bellare and Rogaway 2004; Hall et al. 1998]. A key part of this lemma involves showing that random permutations (RPs) are hard to distinguish from random functions (RFs) by a client (the "adversary") that can only make a bounded number of queries to such functions. For finite sets X and Y, a random function $f : X \longrightarrow Y$ can be sampled by selecting, for each $x \in X$, an independent, uniform sample from Y, to use as the value for $f(x)$. Sometimes it is desirable for f to be invertible (for modeling encryption and decryption of a block cipher). We call such an f a random permutation. The difference between a RP and a RF is then that a RF may have collisions, *i.e.*, values $x_1 \neq x_2$ such that $f(x_1) = f(x_2)$, while a RP never produces collisions.

Consider the following task for an "adversary" $\mathcal{A}$. They are given a function f which may be either a RP or a RF, and their goal is to determine which one they are interacting with by querying f up to Q times and observing the results, *i.e.*, they are not allowed to, say, inspect the code of f. Concretely, $\mathcal{A}$ should return true if it interacts with a RP and false for a RF.

How can $\mathcal{A}$ distinguish the two? If $\mathcal{A}$ finds a collision, then it knows that f cannot be a RP. However, if the domain of f is very large compared to Q, then $\mathcal{A}$ cannot simply search the entire domain for a collision, and its chances of finding a collision are low. Thus, the adversary will have a low chance of correctly distinguishing the two scenarios. The switching lemma makes this formal by showing that the probability that $\mathcal{A}$ returns true for either interaction differs by at most $\frac{Q(Q-1)}{2|\mathsf{dom} f|}$.

Lemma 2.1 (PRP/PRF Switching Lemma). *Let $\mathcal{A}$ be an adversary that asks at most Q queries and let $N = |\mathsf{dom}\, RF| = |\mathsf{dom}\, RP|$. Then*

$$|\Pr[\mathcal{A}(RP) = \mathsf{true}] - \Pr[\mathcal{A}(RF) = \mathsf{true}]| \leq \frac{Q(Q-1)}{2N}.$$

The Switching Lemma gained notoriety because several published proofs of the lemma were found to contain mistakes [Bellare and Rogaway 2004]. This observation was among the motivations for the development of a rigorous framework for cryptographic proofs such as the ones based on "games" [Bellare and Rogaway 2004; Shoup 2004] and the subsequent development of mechanized tools for such proofs [Barthe et al. 2014, 2009; Blanchet 2005].

To simplify the exposition, we first prove in Approxis an instantiation of the lemma with a concrete *weak adversary* $\mathcal{A}_W$, which picks the input for its Q queries $x_0, \ldots, x_{Q-1}$ uniformly at random, without adapting to the response of the queries, and returns the list of outputs:

```
let $\mathcal{A}_W$ $N$ $Q$ $f$ =
    let xys = ref List.empty in
    for $i = 0$ to $(Q - 1)$ do
        let x = rand $(N - 1)$ in
        let y = $f$ x in
        xys ← (x, y) :: ! xys
    ! xys
```

Lemma 2.2 (Weak PRP/PRF Switching Lemma). *Let $\mathcal{A}_W(N, Q, \cdot)$ be the weak adversary defined above, and let $N = |\mathsf{dom}\, RF| = |\mathsf{dom}\, RP|$. Then, for any list of results $\vec{xy}$*

$$|\Pr[\mathcal{A}_W(N, Q, RP) = \vec{xy}] - \Pr[\mathcal{A}_W(N, Q, RF) = \vec{xy}]| \leq \frac{Q(Q-1)}{2N}.$$

We will prove the full Switching Lemma for an arbitrary Q-query adversary later in §6.1.

```
let irf N =                                  let irp N =
    let m = Map.init () in                       let m = Map.init () in
                                                 let l_unused = ref (List.seq 0 N) in
    λ x. if Map.get m x = None then              λ x. if Map.get (! m) x = None then
            let y = rand (N − 1) in                      let len = List.length (! l_unused) in
            Map.set (! m) x y ;                          let k = rand (len − 1) in
                                                         let y = List.nth (! l_unused) k in
                                                         Map.set (! m) x y ;
                                                         l_unused ← remove_nth (! l_unused) y ;
          Map.get m x                                  Map.get m x
```

Fig. 1. Example implementation of idealized RF and RP, parameterized by $N = |\text{dom irf}| = |\text{dom irp}|$.

Random Functions and Permutations in RandML. First we need to model random functions and permutations as programs in RandML. An example implementation of an idealized random function with domain $X = \{0, \ldots, N-1\}$ is given by irf in Figure 1. Upon initialization, irf N creates a reference to an initially empty (finite) map m and returns a function rf. On every call to $rf(x)$, if rf has never been evaluated before on x, a new point $y \in X$ is sampled at random and stored into m. Conversely, if x has been queried before, $rf(x)$ looks up its value in m.

The idealized random permutation irp likewise initializes its internal state and returns a closure rp. However, to sample a new element, rp randomly picks (the index k of) an element y of the list l_unused of values in X that do not yet occur in the codomain of rp, removes y from l_unused, and updates its internal map. Initially, all values in X are unused, but as rp is evaluated on new points, l_unused shrinks. Since each element of X occurs only once in l_unused and gets removed the first time it is picked, rp is guaranteed to remain collision-free.

The Mathematical Intuition. We first give an informal sketch of why the result holds—the formal argument in Approxis will closely mirror this style of reasoning. We want to show that with error probability at most ε, $\mathcal{A}_W(N, Q, \text{irp } N)$ and $\mathcal{A}_W(N, Q, \text{irf } N)$ compute the same list of results xys, where ε is the bound from Lemma 2.2. After initializing the weak adversary, both lists are empty. We claim that the lists remain equal with high probability through each iteration of the for loop. W.l.o.g, we can assume both random samplings of x return the same value. If x has been sampled before, then no new information is gained from the call to f, and the results are equal with the same probability as before. If x is fresh, then irf will sample a new response y out of $\{0, \ldots, N-1\}$, while irp picks an element from l_unused. On the i-th loop iteration, l_unused contains at least $N - i$ elements, and hence the probability of irf sampling an element that *does not* occur in l_unused and hence causes an observable collision is i/N. If irf remains collision free, the probability that both programs compute the same result xys does not change. We can hence establish an upper bound on the probability that $\mathcal{A}_W(N, Q, \text{irp } N)$ and $\mathcal{A}_W(N, Q, \text{irf } N)$ produce different results by summing the probabilities that each loop iteration observes a collision. Since $\sum_{i=0}^{Q-1} i/N = \frac{Q(Q-1)}{2N}$, Lemma 2.2 holds.

The Proof in Approxis. We derive Lemma 2.2 by proving the following pair of refinements.

Lemma 2.3. *Let $Q, N \in \mathbb{N}$, and let $\varepsilon_Q \triangleq \frac{Q(Q-1)}{2N}$. Then*

$$\text{↯}(\varepsilon_Q) \vdash \text{rwp } \mathcal{A}_W(N, q, \text{irp } N) \precsim \mathcal{A}_W(N, Q, \text{irf } N)\ \{x, y.\, x = y\}\,, \text{ and} \tag{1}$$

$$\text{↯}(\varepsilon_Q) \vdash \text{rwp } \mathcal{A}_W(N, Q, \text{irf } N) \precsim \mathcal{A}_W(N, Q, \text{irp } N)\ \{x, y.\, x = y\}\,. \tag{2}$$

We sketch the proof of (1); the other direction is analogous. We prove (1) by reasoning backwards from the conclusion. As a first step, we symbolically evaluate $\mathcal{A}_W(N, q, \text{irp } N)$. Evaluation order forces irp N to evaluate first, which allocates a map m and the list l_unused, and returns a function rp. Afterwards, $\mathcal{A}_W$ allocates a list of results xys. Similarly, evaluating $\mathcal{A}_W(N, q, \mathsf{irf}\ N)$ allocates m′, substitutes rf for f, and then allocates xys′. Note that we will often prove refinements of the style rwp $e_1 \precsim e_2\ \{\Phi\}$ where both e_1 and e_2 manipulate a variable x. We frequently write x and x' for the left-hand side (e_1) and right-hand side (e_2) version of x. Both xys and xys′ point to the empty list at this stage, and the maps vm and vm' are empty. We use the traditional "points-to" connective $p \mapsto v$ from separation logic to say that in the left program, p points to v, and write $p \mapsto_{\mathsf{s}} v$ for the analogous fact about the right program state. We are thus left to prove the following refinement

$$\begin{array}{l}\lightning(\varepsilon_Q)\ *\ \mathsf{xys} \mapsto [\,] * \mathsf{xys}' \mapsto_{\mathsf{s}} [\,] \\ \ *\ \mathit{is\_rp}\ \emptyset\ [0, \dots, N] * \mathit{is\_rf}\ \emptyset\end{array} \vdash \mathsf{rwp}\ (\mathit{loop}_{\mathit{rp}}\ Q\ ;\ !\,\mathsf{xys}) \precsim (\mathit{loop}_{\mathit{rf}}\ Q\ ;\ !\,\mathsf{xys}')\ \{x, y.\, x = y\},$$

where $\mathit{loop}_f\ Q$ stands for the for loop with bound Q, and rp and rf are the functions returned by initializing irp and irf respectively. The proposition $\mathit{is\_rp}\ m\ l$ means that m currently points to the map m and l_unused points to the list l, and $\mathit{is\_rf}\ m$ likewise means that m′ tracks the map m.

We can generalize the goal slightly, and instead show the refinement below. The proof goes by induction on the number i of remaining loop iterations (initially Q):

$$\lightning(\varepsilon_i)\ *\ (\exists m, l.\ \mathit{is\_rp}\ m\ l * \mathsf{len}_i \leq |l| * \mathit{is\_rf}\ m) * \Phi_{\mathit{res}} \mathrel{-\!\!*} \mathsf{rwp}\ (\mathit{loop}_{\mathit{rp}}\ i) \precsim (\mathit{loop}_{\mathit{rf}}\ i)\ \{\Phi_{\mathit{res}}\} \quad (3)$$

$$\text{where } \varepsilon_i \triangleq \textstyle\sum_{k=Q-i}^{Q-1} \frac{k}{N} = \frac{i(2Q-i-1)}{2N}\ ,\ \ \mathsf{len}_i \triangleq N - (Q - i)\ ,\ \ \Phi_{\mathit{res}} \triangleq (\exists \vec{xy}.\ \mathsf{xys} \mapsto \vec{xy} * \mathsf{xys}' \mapsto_{\mathsf{s}} \vec{xy})\,.$$

This maintains the key loop invariant that both rp and rf currently have the same mapping m. The base case $i = 0$ holds, since the loop terminates immediately and the postcondition Φ_{res} holds by assumption. To show the inductive case $i = j + 1$, we have to (1) prove that unrolling the loop once preserves Φ_{res} and then (2) apply the induction hypothesis to the remaining j iterations.

By assumption, we start with $\lightning(\varepsilon_{j+1})$ error credits, which represent the "budget" we can spend on avoiding collisions caused by calls to rf that would result in different results. From an easy calculation if follows that $\varepsilon_{j+1} = \frac{Q-i}{N} + \varepsilon_j$. Our first step is to split this budget resource into two parts $\lightning\left(\frac{Q-i}{N}\right)\ *\ \lightning(\varepsilon_j)$. We will use $\lightning\left(\frac{Q-i}{N}\right)$ to "pay" for avoiding collisions in the current loop iteration and $\lightning(\varepsilon_j)$ to account for the remaining j iterations.

We now focus on the first loop unrolling. The first instruction in the loop body samples an input x from $\{0, \dots, N - 1\}$ that will be used as the next query. Since both programs sample from the same uniform distribution rand $(N - 1)$, we can use the exact coupling rule wp-couple-exact seen earlier to proceed under the assumption that both programs sample the same value n. Next, the programs call $rp\ n$ and $rf\ n$ respectively. If n has been seen before, both functions return the same value. If n is fresh on the other hand, rf samples y from $\{0, \dots, N - 1\}$ whereas rp picks an element from the list of unused values l. By assumption, the length of l is (at least) len_i. The probability that rf produce a collision, *i.e.*, sample an element that is *not* in l, is $\frac{N - \mathsf{len}_i}{N} = \frac{Q-i}{N}$. Since this matches the error credits we have, we can apply the following *approximate* coupling rule of Approxis, which generalizes the simpler version seen earlier:

$$\frac{\overset{\textsc{wp-couple-rand-rand-err-le}}{g : \mathbb{N}_{\leq K} \to \mathbb{N}_{\leq M}\ \text{injection} \qquad \lightning\left(\frac{M-K}{M+1}\right) \qquad K \leq M \qquad \forall k \leq K.\ \mathsf{rwp}\ k \precsim g(k)\ \{\Phi\}}}{\mathsf{rwp}\ \ \mathsf{rand}\, K \precsim \mathsf{rand}\, M\ \{\Phi\}}$$

We instantiate g with $(\lambda\, n.\ \mathsf{List.nth}\ l\ n)$, instantiate M with $N - 1$, and instantiate K with $\mathsf{len}_i - 1 = N - (Q - i) - 1$. By giving up ownership of $\lightning\left(\frac{Q-i}{N}\right)$ for the second premise, we can thus continue the proof under the assumption that the rand $(\mathsf{len} - 1)$ and rand $(N - 1)$ resolve to a pair of values k and $g(k)$ such that $g(k)$, the newly sampled element in rf, is exactly the k-th element of l. Since we assumed Φ_{res} as a hypothesis in Equation (3), and since both programs add the same value $(n, g(k))$ to xys and xys′ respectively, Φ_{res} holds again after the first loop unfolding. We can thus conclude our proof by appealing to the induction hypothesis, paying for the error credit premise with $\lightning(\varepsilon_j)$.

2.2 Error Amplification for Exact Equivalences

As alluded to in the introduction, error credits not only allow us to prove approximate equivalences, they also allow us to prove *exact* equivalences of probabilistic programs. To motivate this, consider the following rejection sampler, where $M < N$.

$$\text{rec sampler } \_ = \text{let } x = \text{rand } N \text{ in if } x \leq M \text{ then } x \text{ else sampler } ()$$

By continuously rejecting samples which do *not* correspond to values in the target set $\{0, \ldots, M\}$ and retrying, sampler eventually produces values that are sampled uniformly from $\{0, \ldots, M\}$. We can state this formally by proving that sampler is equivalent to the expression rand M, *i.e.*, by showing the following two refinements:

$$\begin{aligned} &\text{rwp sampler } () \precsim \text{rand } M \; \{v_1, v_2.\, v_1 = v_2\}, \text{ and} \\ &\text{rwp rand } M \precsim \text{sampler } () \; \{v_1, v_2.\, v_1 = v_2\}. \end{aligned}$$

Perhaps surprisingly, although this kind of equivalence is a standard and important result in randomized algorithms, no existing relational program logic can establish this with couplings, to the best of our knowledge, even with approximate couplings.

To see what goes wrong, let us focus on trying to prove the second refinement, and consider trying to apply a coupling rule to the step where the left program executes rand M and the right executes rand N. If the right program's sample is $\leq M$, then we want that value to be coupled and equal to the value sampled on the left. On the other hand, if the right sample is $> M$, then it will be rejected, so we do not want to couple the result of rand M on the left to this value at all. We only want to couple the left sample to be equal to the eventual later value that ends up being accepted! Trying to use an approximate coupling to force both samples to be equal will not work either, as that would incur a large error, and we are trying to show an exact coupling.

Approxis overcomes this limitation by introducing a new form of couplings called *fragmented* couplings, which can be combined with a technique called *error amplification* introduced by Eris. To start, rather than proving the above refinements with no error credits, we instead merely have to prove them starting with an arbitrarily small positive error credit. That is, we must show:

$$\lightning(\varepsilon) \vdash \text{rwp rand } M \precsim \text{sampler } () \; \{v_1, v_2.\, v_1 = v_2\}$$

for *all* $\varepsilon > 0$. The exact refinement then follows by a limiting argument, as $\varepsilon \longrightarrow 0$, *c.f.*, Corollary 4.2.

But what can we do with an arbitrarily small error credit? After all, it may be too small to apply the intended approximate coupling rule. The solution is that when reasoning about a rand N command, the logic allows us to *amplify* and grow the ε credits along branches of the random outcome, as long as the expected amount of error credit across all branches is still ε. In particular, the following *expectation-preserving* fragmented coupling will allow us to apply this principle.

$$\frac{\begin{array}{cc} & M < N \qquad \lightning(\varepsilon) \qquad \iota \hookrightarrow (M, \varepsilon) \\ \forall m \leq M.\, \text{rwp } m \precsim m \; \{\Phi\} & \forall m > M.\, \iota \hookrightarrow (M, \varepsilon) * \lightning\left(\frac{N+1}{N-M} \cdot \varepsilon\right) \mathrel{-\!\!*} \text{rwp rand } M \; \iota \precsim m \; \{\Phi\} \end{array}}{\text{rwp rand } M \; \iota \precsim \text{rand } N \; \{\Phi\}}$$

Here we present a simplified variant of a more general rule, and the grayed out parts may be ignored for now. We discuss the general rules in §4.1. The rule requires ε credits and lets us relate two sampling operations, rand M and rand N. It asks us to consider two cases: (1) the outcome of the two samplings agree and are within range, and (2) the right sampling is resolved to some

$m > M$ but the left does not sample anything; we also get to assume ownership of $\frac{N+1}{N-M} \cdot \varepsilon$ error credits. We call this a "fragmented" coupling because it allows for one of the coupled programs to not necessarily execute its random sample, depending on what the other program drew.

Applying this rule to the rejection sampler, when the first case occurs, the sample will be accepted and the proof will conclude. In the second case, the rejection sampler will loop. Now, for any starting ε, if we repeatedly increase our error credits by a factor of $\frac{N+1}{N-M}$ by each loop iteration, then eventually we will have a large enough error credit to apply an approximate coupling rule and "force" the right-hand sample to be in range. By doing induction on the number of amplifications needed, we can therefore conclude the proof.

3 Preliminaries

In this section, we recall some basic definitions in probability theory and a notion of *approximate couplings* [Sato 2016]. We then introduce the syntax and semantics of RandML, the language of our programs, and our notion of contextual equivalence.

3.1 Probability Theory

To account for possibly non-terminating behavior of programs, we define our operational semantics using probability *sub*-distributions.

Definition 3.1 (Distribution). *A* discrete subdistribution *(henceforth simply* distribution*) on a countable set A is a function $\mu : A \to [0,1]$ such that $\sum_{a \in A} \mu(a) \leq 1$. The collection of distributions on A is denoted by $\mathcal{D}(A)$.*

Given a predicate P, the *Iverson bracket* $[P]$ evaluates to 1 if P is true and to 0 otherwise.

Lemma 3.2 (Discrete Distribution Monad). *We can equip $\mathcal{D}$ with a monadic structure, with operations*

$$\begin{aligned} &\mathrm{ret}\colon A \to \mathcal{D}(A) && \mathrm{bind}\colon (A \to \mathcal{D}(B)) \to \mathcal{D}(A) \to \mathcal{D}(B) \\ &\mathrm{ret}(a)(a') \triangleq [a = a'] && \mathrm{bind}(f, \mu)(b) \triangleq \sum_{a \in A} \mu(a) \cdot f(a)(b) \end{aligned}$$

We use the notation $\mu \gg\!= f$ for $\mathrm{bind}(f, \mu)$.

Definition 3.3 (Expected value). *Let $\mu \in \mathcal{D}(A)$ be a distribution and $X\colon A \to [0,1]$ a random variable. The* expected value *of X with respect to μ is defined as $\mathbb{E}_\mu[X] \triangleq \sum_{a \in A} \mu(a) \cdot X(a)$.*

Many probabilistic relational program logics use *probabilistic couplings* [Lindvall 2002; Thorisson 2000; Villani 2008], a mathematical tool for reasoning about pairs of probabilistic processes. To reason about approximate equivalence of probabilistic programs, we use a notion of *approximate* probabilistic coupling [Sato 2016].

Definition 3.4 (Approximate Coupling). *Let $\mu_1 \in \mathcal{D}(A)$ and $\mu_2 \in \mathcal{D}(B)$. Given some approximation error $\varepsilon \in [0,1]$ and a relation $R \subseteq A \times B$, we say that there exists an (ε, R)-coupling of μ_1 and μ_2 if for all $[0,1]$-valued random variables $X : A \to [0,1]$ and $Y : B \to [0,1]$, such that $(a, b) \in R$ implies $X(a) \leq Y(b)$, the expected value of X exceeds the expected value of Y by at most ε,* i.e., *$\mathbb{E}_{\mu_1}[X] \leq \mathbb{E}_{\mu_2}[Y] + \varepsilon$. We write $\mu_1 \lesssim_\varepsilon \mu_2 : R$ if an (ε, R)-coupling exists between μ_1 and μ_2.*

Proving existence of (ε, R)-couplings for particular choices of R is useful to prove relations between distributions. When R is the equality relation, couplings can be used to prove bounds on the total variation distance, which has applications when reasoning about convergence properties, as well as in security definitions.

Lemma 3.5. *Let $\mu_1, \mu_2 \in \mathcal{D}(A)$ such that there exists an ε-coupling for the equality relation,* i.e., $\mu_1 \lesssim_\varepsilon \mu_2 : (=)$, *then for all $a \in A$ we have $\mu_1(a) \leq \mu_2(a) + \varepsilon$. If, in addition, $\mu_2 \lesssim_\varepsilon \mu_1 : (=)$, then the total variation distance between μ_1 and μ_2 is at most ε,* i.e., $\sup_{S \subseteq A} |\mu_1(S) - \mu_2(S)| \leq \varepsilon$.

Corollary 3.6. *Let $\mu_1, \mu_2 \in \mathcal{D}(A)$. Then $\mu_1 \lesssim_0 \mu_2 : (=)$ implies that for all $a \in A$, $\mu_1(a) \leq \mu_2(a)$.*

By completeness of the real numbers, we obtain the following limiting theorem.

Lemma 3.7. *Let $\mu_1, \mu_2 \in \mathcal{D}(A)$ and $\varepsilon \in [0, 1]$. If $\mu_1 \lesssim_{\varepsilon'} \mu_2 : R$ for all $\varepsilon' > \varepsilon$ then $\mu_1 \lesssim_\varepsilon \mu_2 : R$.*

To construct couplings between program executions, we can compose couplings of single steps of executions. This is possible because couplings compose along the bind of the distribution monad. Let $\mu_1 \in \mathcal{D}(A)$, $\mu_2 \in \mathcal{D}(B)$, $f : A \to \mathcal{D}(A')$, $g : B \to \mathcal{D}(B')$, $R \subseteq A \times B$, and $R' \subseteq A' \times B'$.

Lemma 3.8. *If $\mu_1 \lesssim_\varepsilon \mu_2 : R$ and $\forall (a, b) \in R, f(a) \lesssim_{\varepsilon'} g(b) : R'$, then $(\mu_1 \gg\!= f) \lesssim_{\varepsilon+\varepsilon'} (\mu_2 \gg\!= g) : R'$.*

We can strengthen this lemma by letting the grading ε' for the continuation vary depending on the value that a takes, and consider its expected value w.r.t. μ_1 when composing the couplings:

Lemma 3.9. *Let $\mathcal{E} : A \to [0, 1]$. If $\mu_1 \lesssim_\varepsilon \mu_2 : R$ and $\forall (a, b) \in R, f(a) \lesssim_{\mathcal{E}(a)} g(b) : R'$, then $(\mu_1 \gg\!= f) \lesssim_{\varepsilon+\varepsilon'} (\mu_2 \gg\!= g) : R'$ where $\varepsilon' = \mathbb{E}_{\mu_1}[\mathcal{E}]$.*

Symmetrically, we can vary the error on B and consider its expected value w.r.t. μ_2:

Lemma 3.10. *Let $\mathcal{E} : B \to [0, 1]$. If $\mu_1 \lesssim_\varepsilon \mu_2 : R$ and $\forall (a, b) \in R, f(a) \lesssim_{\mathcal{E}(b)} g(b) : R'$, then $(\mu_1 \gg\!= f) \lesssim_{\varepsilon+\varepsilon'} (\mu_2 \gg\!= g) : R'$ where $\varepsilon' = \mathbb{E}_{\mu_2}[\mathcal{E}]$.*

To the best of our knowledge, the *expectation-preserving composition lemmas* 3.9 and 3.10 are novel, at least in the context of program logics. We apply these results for rules such as the expectation-preserving fragmented coupling rule presented in §2.2, where we can amplify the error credit for certain branches as long as the expected amount of error credit across all branches remains the same.

3.2 The RandML Language and Operational Semantics

The RandML language that we consider is an ML-like language with probabilistic uniform sampling, higher-order functions, higher-order state, recursive types, and impredicative type polymorphism.

The syntax is defined by the grammar below.

$$
\begin{aligned}
v, w \in \mathit{Val} &::= z \in \mathbb{Z} \mid b \in \mathbb{B} \mid () \mid \ell \in \mathit{Loc} \mid \mathsf{rec}\,\mathsf{f}\,\mathsf{x} = e \mid (v, w) \mid \mathsf{inl}\, v \mid \mathsf{inr}\, v \\
e \in \mathit{Expr} &::= v \mid \mathsf{x} \mid \mathsf{rec}\,\mathsf{f}\,\mathsf{x} = e \mid e_1\, e_2 \mid e_1 + e_2 \mid e_1 - e_2 \mid \ldots \mid \mathsf{if}\, e\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 \mid (e_1, e_2) \mid \mathsf{fst}\, e \mid \ldots \\
&\quad\ \mathsf{ref}\ e_1 \mid !\, e \mid e_1 \leftarrow e_2 \mid e_1[e_2] \mid \mathsf{rand}\ e \mid \mathsf{pack}\, e \mid \mathsf{unpack}\, e\, \mathsf{as}\, x\, \mathsf{in}\, e \mid \ldots \\
K \in \mathit{Ectx} &::= - \mid e\, K \mid K\, v \mid \mathsf{ref}\ K \mid !\, K \mid e \leftarrow K \mid K \leftarrow v \mid \mathsf{rand}\, K \mid \ldots \\
\sigma \in \mathit{State} &\triangleq \mathit{Loc} \xrightarrow{\text{fin}} \mathit{Val} \qquad \rho \in \mathit{Cfg} \triangleq \mathit{Expr} \times \mathit{State} \\
\tau \in \mathit{Type} &::= \alpha \mid \mathsf{unit} \mid \mathsf{bool} \mid \mathsf{nat} \mid \mathsf{int} \mid \tau \times \tau \mid \tau + \tau \mid \tau \to \tau \mid \forall \alpha.\, \tau \mid \exists \alpha.\, \tau \mid \mu\, \alpha.\, \tau \mid \mathsf{ref}\ \tau
\end{aligned}
$$

The term language is mostly standard. We use $\mathsf{ref}\, e_1$ to allocate a new reference containing the value returned by e_1, $!\, e$ to dereference the location e evaluates to, and $e_1 \leftarrow e_2$ to evaluate e_2 and assign the result to the location that e_1 evaluates to. We often refer to a recursive function value $\mathsf{rec}\,\mathsf{f}\,\mathsf{x} = e$ by its name f. The operation $\mathsf{rand}\ N$ denotes uniform random sampling over $\{0, \ldots, N\}$.

Finally, we have several terms related to typing operations *e.g.*, $\mathsf{pack}\, e$ and $\mathsf{unpack}\, e_1\, \mathsf{as}\, x\, \mathsf{in}\, e_2$ are used for introducing and eliminating existential types. We write $\Theta \mid \Gamma \vdash e\colon \tau$ to denote that e has type τ in the typing context $\Theta \mid \Gamma$, which consists of a context of type variables Θ and a context of program variables Γ. The inference rules for the typing judgments are standard (see, *e.g.*, Frumin et al. [2021] or the Coq formalization).

Operational Semantics. To define program execution, we define $\mathsf{step}(\rho) \in \mathcal{D}(\mathit{Cfg})$, the distribution induced by the single step reduction of configuration $\rho \in \mathit{Cfg}$. The semantics is mostly standard. We first define head reductions and then lift it to reduction in an evaluation context K. All non-probabilistic constructs reduce deterministically as usual, *e.g.*, $\mathsf{step}((\lambda\, x.\, e)\, v, \sigma) = \mathsf{ret}(e[v/x], \sigma)$. We write $e \overset{\text{pure}}{\rightsquigarrow} e'$ if the evaluation is deterministic and holds independently of the state, *e.g.*, $(\lambda\, x.\, e)\, v \overset{\text{pure}}{\rightsquigarrow} e[v/x]$ and $\mathsf{fst}(v_1, v_2) \overset{\text{pure}}{\rightsquigarrow} v_1$. The probabilistic choice $\mathsf{rand}\, N$ reduces uniformly at random, *i.e.*,

$$\mathsf{step}(\mathsf{rand}\, N, \sigma)(n, \sigma) \triangleq \begin{cases} \frac{1}{N+1} & \text{for } n \in \{0, 1, \ldots, N\}, \\ 0 & \text{otherwise.} \end{cases}$$

With the single step reduction $\mathsf{step}(-,-)$ defined, we next define a step-stratified execution probability $\mathsf{exec}_n \colon \mathit{Cfg} \to \mathcal{D}(\mathit{Val})$ by induction on n:

$$\mathsf{exec}_0(e, \sigma)(v) \triangleq \begin{cases} 1 & \text{if } e \in \mathit{Val} \land e = v, \\ 0 & \text{otherwise.} \end{cases}$$

$$\mathsf{exec}_{m+1}(e, \sigma)(v) \triangleq \begin{cases} 1 & \text{if } e \in \mathit{Val} \land e = v, \\ \sum_{(e',\sigma') \in \mathit{Expr} \times \mathit{State}} \mathsf{step}(e, \sigma)(e', \sigma') \cdot \mathsf{exec}_m(e', \sigma')(v) & \text{otherwise.} \end{cases}$$

That is, $\mathsf{exec}_n(e, \sigma)(v)$ is the probability of stepping from the configuration (e, σ) to a value v in less than n steps. The probability that a execution, starting from configuration ρ, reaches a value v is taken as the limit of its stratified approximations, which exists by monotonicity and boundedness:

$$\mathsf{exec}(\rho)(v) \triangleq \mathsf{lim}_{n \to \infty} \mathsf{exec}_n(\rho)(v)$$

The termination probability of an execution from configuration ρ is $\mathsf{exec}_{\Downarrow}(\rho) \triangleq \Sigma_{v \in \mathit{Val}} \mathsf{exec}(\rho)(v)$.

The definition of program execution as a distribution leads to a natural notion of ε-approximation. We say that e_1 *ε-approximates* e_2 if $\mathsf{exec}(e_1, \sigma)(v) \leq \mathsf{exec}(e_2, \sigma)(v) + \varepsilon$ for all v, σ. By Lemma 3.5, we can show such approximations by establishing an approximate coupling of the executions of e_1 and e_2. We say e_1 and e_2 are ε-equivalent if both e_1 ε-approximates e_2 and e_2 ε-approximates e_1. In that case, we have $|\mathsf{exec}(e_1, \sigma)(v) - \mathsf{exec}(e_2, \sigma)(v)| \leq \varepsilon$ for all v, σ.

Example 3.11. Consider the program below

$$e \triangleq \mathsf{let}\, x = \mathsf{rand}\, N\, \mathsf{in}\, x \leq M$$

and assume $M \leq N$. Evaluation order dictates that the random sampling is resolved first. Then we have, for any $\sigma \colon \mathit{State}$ and any $0 \leq n \leq N$:

$$\mathsf{step}(e, \sigma)(\mathsf{let}\, x = n\, \mathsf{in}\, x \leq M, \sigma) = \frac{1}{N+1}$$

The probability of stepping to any other configuration is 0. Fixing a particular n, the next step is deterministic, and therefore we have

$$\mathsf{step}(\mathsf{let}\, x = n\, \mathsf{in}\, x \leq M, \sigma) = \mathsf{ret}(n \leq M, \sigma)$$

The final step is also deterministic and just evaluates the inequality $n \leq M$ to either true or false. Collecting all of the probabilities of the succesful comparisons together, we can show

$$\mathsf{exec}_3(e, \sigma)(\mathsf{true}) = \frac{M+1}{N+1}$$

and trivially at the limit

$$\mathsf{exec}(e, \sigma)(\mathsf{true}) = \frac{M+1}{N+1}$$

Note that, since the execution of e takes exactly 3 steps to reach to a value, executing e for fewer steps returns the zero distribution on values:

$$\mathrm{exec}_0(e,\sigma) = \mathrm{exec}_1(e,\sigma) = \mathrm{exec}_2(e,\sigma) = \lambda v.0$$

Presampling Tapes. Standard probabilistic coupling logics require aligning or "synchronizing" sampling statements of the two programs under consideration. For example, both programs have to be executing the sample statements we want to couple for their next step when applying a coupling rule. However, it is not always possible to synchronize sampling statements in this way, especially for higher-order programs. To address this issue, Gregersen et al. [2024] introduce *asynchronous coupling*. As we will see, the same mechanisms are useful for approximate relational reasoning.

Asynchronous couplings are introduced through dynamically-allocated *presampling tapes* that are added to the language. Intuitively, presampling tapes will allow us *in the logic* to presample (and in turn couple) the outcome of future sampling statements. Formally, presampling tapes appear as two new constructs added to the programming language.

$$\begin{aligned}
v \in \mathit{Val} &::= \dots \mid \iota \in \mathit{Label} & \sigma \in \mathit{State} &\triangleq (\mathit{Loc} \xrightarrow{\text{fin}} \mathit{Val}) \times (\mathit{Label} \xrightarrow{\text{fin}} \mathit{Tape}) \\
e \in \mathit{Expr} &::= \dots \mid \mathsf{tape}\ e \mid \mathsf{rand}\ e_1\ e_2 & t \in \mathit{Tape} &\triangleq \{(N, \vec{n}) \mid N \in \mathbb{N} \wedge \vec{n} \in \mathbb{N}^*_{\leq N}\} \\
K \in \mathit{Ectx} &::= \dots \mid \mathsf{tape}\ K \mid \mathsf{rand}\ e\ K \mid \mathsf{rand}\ K\ v & \tau \in \mathit{Type} &::= \dots \mid \mathsf{tape}
\end{aligned}$$

The $\mathsf{tape}\,N$ operation allocates a new fresh tape with label ι and upper bound N, representing future outcomes of $\mathsf{rand}\,N\,\iota$ operations. The rand primitive can now (optionally) be annotated with the tape label ι. If the corresponding tape is empty, $\mathsf{rand}\,N\,\iota$ reduces to any $n \leq N$ with equal probability, just as if it had not been labeled. But if the tape is *not* empty, then $\mathsf{rand}\,N\,\iota$ reduces *deterministically* by taking off the first element of the tape and returning it.

$$\mathrm{step}(\mathsf{tape}\,N, \sigma) \triangleq \begin{cases} \mathrm{ret}(\iota, \sigma[\iota \mapsto (N, \epsilon)]) & \iota = \mathrm{fresh}(\sigma),\ N \geq 0\,, \\ \mathrm{ret}(\iota, \sigma[\iota \mapsto (0, \epsilon)]) & \iota = \mathrm{fresh}(\sigma),\ N < 0\,. \end{cases}$$

$$\mathrm{step}(\mathsf{rand}\,N\,\iota, \sigma[\iota \mapsto (N,\epsilon)])(n, \sigma[\iota \mapsto (N,\epsilon)]) \triangleq \begin{cases} \frac{1}{N+1} & \text{for } n \in \{0, 1, \dots, N\}, \\ 0 & \text{otherwise.} \end{cases}$$

$$\mathrm{step}(\mathsf{rand}\,N\,\iota, \sigma[\iota \mapsto (N, n :: w)]) \triangleq \mathrm{ret}(n, \sigma[\iota \mapsto (N, w)])$$

Note that *no* primitives in the language add values to the tapes. Instead, values are added to tapes as part of presampling steps that will be *ghost operations* appearing only in the logic. In fact, labeled and unlabeled sampling operations are contextually equivalent [Gregersen et al. 2024]. This result follows from the fact that the ghost operations for adding values to tapes are *erasable* in the following sense:

Definition 3.12 (Erasable). *Let $\mu \in \mathcal{D}(\mathit{State})$ and $\sigma \in \mathit{State}$.*

$$\mathrm{erasable}(\mu, \sigma) \triangleq \forall e, n.\ \mathrm{exec}_n(e,\sigma) = (\mu \gg\!= \lambda\,\sigma'.\,\mathrm{exec}_n(e,\sigma'))$$

Erasability of μ w.r.t. σ intuitively captures that distribution μ does not influence the probabilistic *outcome* of any program execution from state σ. For example, erasable(ret(σ), σ) trivially holds by the left identity law of the distribution monad. More interestingly, in RandML, erasable($\mathrm{sstep}_\iota(\sigma), \sigma$) holds where $\mathrm{sstep}_\iota(\sigma)$ is the distribution of the ghost operation that samples a fresh value uniformly onto the end of the presampling tape with label ι in state σ. This is the essence of the soundness of asynchronous couplings and ultimately what allows us to validate rules such as wp-tape-tape-append and wp-many-to-one, which we explain later in §4.1.

3.3 Contextual Refinement and Equivalence

A program context C is an expression with a hole and we write $C[e]$ for the term resulting from replacing the hole in C by e. Contexts are also typed; we write $(C \colon (\Theta \mid \Gamma \vdash \tau) \Rightarrow (\Theta' \mid \Gamma' \vdash \tau'))$ whenever $\Theta' \mid \Gamma' \vdash C[e] \colon \tau'$ for every well-typed $\Theta \mid \Gamma \vdash e \colon \tau$.

The notion of contextual refinement that we use is standard and uses the termination probability $\mathsf{exec}_{\Downarrow}$ as observation predicate. We say expression e_1 *contextually refines* expression e_2 if for all well-typed program contexts C resulting in a closed program then the termination probability of $C[e_1]$ is bounded by the termination probability of $C[e_2]$:

$$\Theta \mid \Gamma \vdash e_1 \precsim_{\mathrm{ctx}} e_2 : \tau \triangleq \forall \tau', (C : (\Theta \mid \Gamma \vdash \tau) \Rightarrow (\emptyset \mid \emptyset \vdash \tau')), \sigma.\ \mathsf{exec}_{\Downarrow}(C[e_1], \sigma) \leq \mathsf{exec}_{\Downarrow}(C[e_2], \sigma)$$

Note that contextual refinement is a precongruence, and that the statement itself is in the meta-logic (*e.g.*, Coq) and makes no mention of Approxis or Iris. We define *contextual equivalence* $\Theta \mid \Gamma \vdash e_1 \simeq_{\mathrm{ctx}} e_2 : \tau$ as refinement in both directions, *i.e.*, $\Theta \mid \Gamma \vdash e_1 \precsim_{\mathrm{ctx}} e_2 : \tau$ and $\Theta \mid \Gamma \vdash e_2 \precsim_{\mathrm{ctx}} e_1 : \tau$.

4 An Approximate Relational Logic

In this section, we introduce the relational Approxis logic and its soundness theorem, with an emphasis on the novel relational rules that interact with error credits or with presampling tapes. We then demonstrate the logic on a simple rejection sampler.

Approxis is built on top of the Iris separation logic framework [Jung et al. 2018] and hence inherits many of Iris's logical connectives. A selection of Approxis propositions is shown below.

$$\begin{aligned} P, Q \in \mathit{iProp} ::= {} & \mathsf{True} \mid \mathsf{False} \mid P \land Q \mid P \lor Q \mid P \Rightarrow Q \mid \forall x.\, P \mid \exists x.\, P \mid P \ast Q \mid P \mathrel{-\!\!\ast} Q \mid \\ & \ell \mapsto v \mid \ell \mapsto_{\mathsf{s}} v \mid \iota \hookrightarrow (N, \vec{n}) \mid \iota \hookrightarrow_{\mathsf{s}} (N, \vec{n}) \mid \lightning(\varepsilon) \mid \mathsf{rwp}\ e_1 \precsim e_2\ \{v_1, v_2.\, P\} \mid \ldots \end{aligned}$$

Most of the propositions are standard, such as separating conjunction $P \ast Q$ and separating implication $P \mathrel{-\!\!\ast} Q$ (the magic wand). As we saw earlier, the heap points-to assertion that denotes ownership of location ℓ comes in two forms: $\ell \mapsto v$ for the left-hand side, and $\ell \mapsto_{\mathsf{s}} v$ for the right-hand side (the "specification" side). Similarly, since presampling tapes are part of the state, we also have tape points-to assertions for both sides: $\iota \hookrightarrow (N, \vec{n})$ and $\iota \hookrightarrow_{\mathsf{s}} (N, \vec{n})$, respectively.

Inspired by Eris [Aguirre et al. 2024], we interpret errors as resources in our logic using the $\lightning(\varepsilon)$ connective for $\varepsilon \in [0, 1]$. Intuitively, $\lightning(\varepsilon)$ denotes ownership of ε error credits that can be spent to do ε-approximate reasoning. Error credits can be split and combined, *i.e.*, $\lightning(\varepsilon_1 + \varepsilon_2) \dashv\vdash \lightning(\varepsilon_1) \ast \lightning(\varepsilon_2)$. Another important fact is that ownership of 1 error credit immediately leads to a contradiction, *i.e.*, $\lightning(1) \vdash \mathsf{False}$. Intuitively, this is sound because there always exists a trivial $(1, \varphi)$-coupling for any two distributions and for any φ.

To show that e_1 ε-approximates e_2 we prove an entailment of the form $\lightning(\varepsilon) \vdash \mathsf{rwp}\ e_1 \precsim e_2\ \{v, v'.v = v'\}$. The following soundness theorem says we may then conclude the existence of an ε-approximate coupling under the equality relation.

Theorem 4.1 (Adequacy). *Let $\varphi \subseteq \mathit{Val} \times \mathit{Val}$ be a relation and $\varepsilon \in [0, 1]$. If $\lightning(\varepsilon) \vdash \mathsf{rwp}\ e_1 \precsim e_2\ \{\varphi\}$ then $\mathsf{exec}(e_1, \sigma_1) \precsim_{\varepsilon} \mathsf{exec}(e_2, \sigma_2) : \varphi$ for all σ_1 and σ_2.*

The result is stated here to provide the reader with a semantic understanding of the rules we will present in this section. In §7 we will explain the underlying model in more detail and discuss the proof of this result.

As a corollary of the above, we get the following *error-limiting* result by appealing to Lemma 3.7.

Corollary 4.2 (Error-Limiting Adequacy). *Let $\varphi \subseteq \mathit{Val} \times \mathit{Val}$ be a relation and $\varepsilon \in [0, 1]$. If $\lightning(\varepsilon') \vdash \mathsf{rwp}\ e_1 \precsim e_2\ \{\varphi\}$ for all $\varepsilon' > \varepsilon$ then $\mathsf{exec}(e_1, \sigma_1) \precsim_{\varepsilon} \mathsf{exec}(e_2, \sigma_2) : \varphi$ for all σ_1 and σ_2.*

The corollary is similar to Theorem 4.1 in that we obtain an approximate (ε, φ)-coupling of the execution of the two programs. However, instead of establishing the weakest precondition given ε credits, one has to prove the weakest precondition given ε' credits for an arbitrary $\varepsilon' > \varepsilon$. Note that by picking ε to be 0 this also allows us to establish exact equivalences as we demonstrate in §4.2.

4.1 Rules of Approxis

In this section, we present a selection of the rules of Approxis. The rules are categorized into four classes. We start with program-logic rules that are the relatively standard laws that most relational separation logics enjoy. We then discuss of (approximate and non-approximate) coupling rules. Afterwards, we consider rules that use presampling tapes to reason about more complicated couplings. We conclude with a discussion of the error amplification proof technique which Approxis supports for reasoning about recursive programs.

Program-Logic Rules. We note that most rules (except where explicitly mentioned, in particular wp-rec) have both left- and right-sided variants. For brevity, we present only left-sided variants, right-sided variants are symmetric and use specification-side connectives ($\mapsto_{\mathrm{s}}$ and $\hookrightarrow_{\mathrm{s}}$).

Although Approxis is a separation logic for reasoning about probabilistic programs, the rules of the non-probabilistic fragment are identical to the structural and computational rules found in most logics for non-probabilistic programs. A selection of rules for the deterministic fragment is found in Figure 2. For example, Approxis satisfies the relational bind rule (wp-bind), rules for symbolically taking deterministic "pure" steps—steps that do not depend on state (wp-pure-l), and rules for interacting with the heap (wp-load-l). The rule wp-rec is the standard recursive function rule found in general program logics. (We do not have a "standard" recursive function rule for the right-hand side program because of how our refinement weakest-precondition assertion is defined. See §7 for more details.)

$$\frac{\mathsf{rwp}\ e_1 \precsim e_2\ \{\Psi\} \qquad \forall v_1, v_2.\ \Psi(v_1, v_2) \mathrel{-\!\!*} \mathsf{rwp}\ K[v_1] \precsim K[v_2]\ \{\Phi\}}{\mathsf{rwp}\ K[e_1] \precsim K'[e_2]\ \{\Phi\}}\ \textsc{wp-bind}$$

$$\frac{e_1 \overset{\text{pure}}{\rightsquigarrow} e_1' \qquad \mathsf{rwp}\ e_1' \precsim e_2\ \{\Phi\}}{\mathsf{rwp}\ e_1 \precsim e_2\ \{\Phi\}}\ \textsc{wp-pure-l} \qquad\qquad \frac{\ell \mapsto v \qquad \ell \mapsto v \mathrel{-\!\!*} \mathsf{rwp}\ v \precsim e\ \{\Phi\}}{\mathsf{rwp}\ {!\ell} \precsim e\ \{\Phi\}}\ \textsc{wp-load-l}$$

$$\frac{(\forall w.\ \mathsf{rwp}\ (\mathsf{rec}\ f\ x = e)\ w \precsim e'\ \{\Phi\}) \vdash \mathsf{rwp}\ e_1[v/x][(\mathsf{rec}\ f\ x = e)/f] \precsim e'\ \{\Phi\}}{\vdash \mathsf{rwp}\ (\mathsf{rec}\ f\ x = e)\ v \precsim e'\ \{\Phi\}}\ \textsc{wp-rec}$$

Fig. 2. A selection of the deterministic program-logic rules of Approxis.

The program-logic rules for the probabilistic fragment of RandML and presampling tapes are shown in Figure 3. These rules reflect the operational semantics of RandML. For situations where we only want to progress the left program's random sampling without coupling, the rule wp-rand-l can be used. The rule wp-alloc-tape-l allocates a fresh tape and returns its label. The rules for sampling from a tape ι depend on the contents of the tape: if the tape is *not* empty, we pop and return the first value (wp-rand-tape-l). If the tape is empty, we sample an arbitrary integer from 0 to N (wp-rand-tape-empty-l), just as for rand N without a tape annotation.

We emphasize that all of the rules shown so far are also found in the relational logic of Clutch [Gregersen et al. 2024], *except* that the right-hand-side rules in Clutch require that the left-hand-side program is *not* a value. This side-condition is a limitation of the model of Clutch that we eliminate in Approxis. (This seemingly small improvement required *significant* changes to the model, which we detail in §7.)

$$\frac{\forall n \le N.\, \mathsf{rwp}\ n \precsim e\ \{\Phi\}}{\mathsf{rwp}\ \ \mathsf{rand}\, N \precsim e\ \{\Phi\}}\ \textsc{wp-rand-l} \qquad \frac{\forall \iota.\, \iota \hookrightarrow (N, \epsilon) \mathrel{-\!\!*} \mathsf{rwp}\ \iota \precsim e\ \{\Phi\}}{\mathsf{rwp}\ \ \mathsf{tape}\, N \precsim e\ \{\Phi\}}\ \textsc{wp-alloc-tape-l}$$

$$\frac{\iota \hookrightarrow (N, n \cdot \vec{n}) \qquad \iota \hookrightarrow (N, \vec{n}) \mathrel{-\!\!*} \mathsf{rwp}\ n \precsim e\ \{\Phi\}}{\mathsf{rwp}\ \ \mathsf{rand}\, N\ \iota \precsim e\ \{\Phi\}}\ \textsc{wp-rand-tape-l}$$

$$\frac{\iota \hookrightarrow (N, \epsilon) \qquad \forall n \le N.\, \iota \hookrightarrow (N, \epsilon) \mathrel{-\!\!*} \mathsf{rwp}\ n \precsim e\ \{\Phi\}}{\mathsf{rwp}\ \ \mathsf{rand}\, N\ \iota \precsim e\ \{\Phi\}}\ \textsc{wp-rand-tape-empty-l}$$

Fig. 3. A selection of program-logic rules of Approxis for the probabilistic operations.

Approximate Coupling Rules. The rules shown so far allow one to symbolically progress either the left- or right-hand side program of the weakest precondition assertion, independently of each other. However to prove interesting relational properties, we need to progress the programs in a related manner using coupling rules, which we saw special cases of in §2.

First, we have wp-couple-rand-rand-err-le which relates sampling $\mathsf{rand}\, N$ with $\mathsf{rand}\, M$ where $N \le M$. Here $f : \mathbb{N}_{\le N} \to \mathbb{N}_{\le M}$ is an injective function ($\mathbb{N}_{\le N}$ denotes the natural numbers $\le N$) and by spending $\frac{M-N}{M+1}$ error credits, we may continue reasoning as if the return values are "synchronized" and related by f. This is also the rule we used for proving the switching lemma presented in §2. Note that in the special case where $N = M$, this generalizes the traditional coupling rule found in exact coupling logics (*e.g.* Clutch [Gregersen et al. 2024]) where no error is incurred; the wp-couple-exact rule in §2 is an example of this special case.

The wp-couple-rand-rand-err-ge rule works almost identically, except that the inequality of the bound is reversed, *i.e.*, $N \ge M$. (We mention that all the other rules we present in this paper have symmetric versions, just as for wp-couple-rand-rand-err-le and wp-couple-rand-rand-err-ge. For the sake of brevity, we shall only present one direction of each pair of rules subsequently.)

wp-couple-rand-rand-err-le

$$\frac{f : \mathbb{N}_{\le N} \to \mathbb{N}_{\le M} \text{ injection} \qquad \lightning\left(\tfrac{M-N}{M+1}\right) \qquad N \le M \qquad \forall n \le N.\, \mathsf{rwp}\ n \precsim f(n)\ \{\Phi\}}{\mathsf{rwp}\ \ \mathsf{rand}\, N \precsim \mathsf{rand}\, M\ \{\Phi\}}$$

wp-couple-rand-rand-err-ge

$$\frac{f : \mathbb{N}_{\le M} \to \mathbb{N}_{\le N} \text{ injection} \qquad \lightning\left(\tfrac{N-M}{N+1}\right) \qquad N \ge M \qquad \forall n \le M.\, \mathsf{rwp}\ f(n) \precsim n\ \{\Phi\}}{\mathsf{rwp}\ \ \mathsf{rand}\, N \precsim \mathsf{rand}\, M\ \{\Phi\}}$$

As in Clutch, Approxis also supports *asynchronous* coupling. For example, the wp-couple-tape-tape-err-ge rule below is a variant of wp-couple-rand-rand-err-ge where, instead of two program samplings, we couple two tape samplings.

wp-couple-tape-tape-err-ge

$$\frac{\begin{array}{c} f : \mathbb{N}_{\le M} \to \mathbb{N}_{\le N} \text{ injection} \qquad N \ge M \qquad \iota \hookrightarrow (N, \vec{n}) \qquad \iota' \hookrightarrow_{\mathsf{s}} (M, \vec{m}) \\ \lightning\left(\tfrac{N-M}{N+1}\right) \qquad \forall n \le M.\, \iota \hookrightarrow (N, \vec{n} \cdot f(n)) * \iota' \hookrightarrow_{\mathsf{s}} (M, \vec{m} \cdot n) \mathrel{-\!\!*} \mathsf{rwp}\ e_1 \precsim e_2\ \{\Phi\} \end{array}}{\mathsf{rwp}\ e_1 \precsim e_2\ \{\Phi\}}$$

Many-to-One and Fragmented Coupling Rules. The coupling rules shown so far allow one to couple one sampling with another. However, in certain cases, we may need to couple one sampling to *zero* or *multiple* possibly-non-adjacent samplings. Consider the following two programs as an example: $e_1 \triangleq 2 \cdot \mathsf{rand}\ 1 + \mathsf{rand}\ 1$ and $e_2 \triangleq \mathsf{rand}\ 3$. These programs are equivalent: e_1 samples two bits and returns the result interpreted in base 2, while e_2 samples directly from the same distribution. None of the coupling rules shown so far would allow us to relate these two programs.

Presampling tapes turn out to be a succinct and uniform solution to this problem, as smoothly enabled by the new and more flexible model of Approxis. By reasoning about values stored in tapes, we can construct more intricate couplings that do not adhere to the one-to-one pattern exhibited in our previous rules. This notion is captured by the following general rule:

$$\frac{\begin{array}{c}\textsc{wp-tape-tape-append} \\ ⚡(\varepsilon) \qquad \iota \hookrightarrow (N, \vec{n}) \qquad \iota' \hookrightarrow_{\mathsf{s}} (M, \vec{m}) \qquad \mathsf{unif}(N+1)^p \lesssim_\varepsilon \mathsf{unif}(M+1)^q : R \\ \forall (v, w) \in R.\ \iota \hookrightarrow (N, \vec{n} +\!\!+ v) * \iota' \hookrightarrow_{\mathsf{s}} (M, \vec{m} +\!\!+ w) -\!\!* \mathsf{rwp}\ e_1 \precsim e_2\ \{\Phi\}\end{array}}{\mathsf{rwp}\ e_1 \precsim e_2\ \{\Phi\}}$$

Here $\mathsf{unif}(x)^y$ refers to the uniform distribution of lists in $\mathsf{List}(x, y)$, where $\mathsf{List}(x, y)$ denotes lists of length y containing integers not larger than x. Assume we want to prove $\mathsf{rwp}\ e_1 \precsim e_2\ \{\Phi\}$, and we are given $⚡(\varepsilon)$ error credits and tapes ι and ι' of bounds N and M on the left and right side of the refinement, respectively. Then the rule says it suffices to (1) choose two lengths p and q, and a relation R over $\mathsf{List}(N, p)$ and $\mathsf{List}(M, q)$, (2) prove an approximate coupling $\mathsf{unif}(N+1)^p \lesssim_\varepsilon \mathsf{unif}(M+1)^q : R$, and (3) show for all lists $(v, w) \in R$, the refinement weakest-precondition assertion holds after v and w are appended to some tapes ι and ι', respectively.

Intuitively, wp-tape-tape-append is sound because appending lists sampled from the distribution $\mathsf{unif}(x)^y$ is an erasable action (see Definition 3.12), meaning that it does not influence the probabilistic execution of any program. However, proving the asynchronous coupling $\mu_1 \lesssim_{\varepsilon_1} \mu_2 : R$ for some arbitrary μ_1, μ_2, and R is generally not an easy task, so we provide various rules which are special instances of wp-tape-tape-append.

First, we introduce the wp-many-to-one rule, derived from wp-tape-tape-append, that allows us to couple one sampling onto a tape with *multiple* samplings onto another tape. This allows us to handle the two programs e_1 and e_2 above that generate samples from $\{0, 1, 2, 3\}$.

$$\frac{\begin{array}{c}\textsc{wp-many-to-one} \\ (N+1)^p = M+1 \qquad \iota \hookrightarrow (N, \vec{n}) \qquad \iota' \hookrightarrow_{\mathsf{s}} (M, \vec{m}) \\ \forall l.\ \mathsf{length}(l) = p * \iota \hookrightarrow (N, \vec{n} +\!\!+ l) * \iota' \hookrightarrow_{\mathsf{s}} (M, \vec{m} \cdot \mathsf{decoder}(N, l)) -\!\!* \mathsf{rwp}\ e_1 \precsim e_2\ \{\Phi\}\end{array}}{\mathsf{rwp}\ e_1 \precsim e_2\ \{\Phi\}}$$

The meta-level function decoder takes as arguments an integer N and a list of integers l whose elements are smaller than or equal to N, and returns the integer represented by the list l in base $N+1$. For example $\mathsf{decoder}(1, [1, 1, 0])$ returns the value 6. Intuitively, given that $(N+1)^p = M+1$, wp-many-to-one couples p samplings onto the tape ι with a single sampling onto ι', such that they are related by the decoder function.

In addition to many-to-one couplings, Approxis also introduces a class of *fragmented* coupling rules, which we briefly introduced in §2.2. Fragmented coupling is a novel kind of coupling rule where the number of values inserted into the tapes are not uniform for all possible branches. Fragmented coupling rule are derived from a stronger notion of wp-tape-tape-append, but the underlying principle is the same, in that the actions of inserting lists to various tapes are erasable and do not affect the probabilistic outcomes of program execution. The notion of fragmented couplings is captured by the following rule WP-FRAGMENTED-R-EXP:

$$\textsc{wp-fragmented-r-exp}\quad \dfrac{\begin{array}{c} f : \mathbb{N}_{\leq N} \to \mathbb{N}_{\leq M} \text{ injection} \qquad N < M \qquad \lightning(\varepsilon) \qquad \iota \hookrightarrow (N, \vec{n}) \qquad \iota' \hookrightarrow_{\mathsf{s}} (M, \vec{m}) \\ \forall m \leq M.\, \iota' \hookrightarrow_{\mathsf{s}} (M, \vec{m} \cdot m) \ast \left(\text{if } m \in \mathsf{img}(f) \begin{array}{l} \text{then } \iota \hookrightarrow (N, \vec{n} \cdot f^{-1}(m)) \\ \text{else } \ \iota \hookrightarrow (N, \vec{n}) \ast \lightning\left(\frac{M+1}{M-N} \cdot \varepsilon\right) \end{array} \right) \mathrel{-\!\!*} \mathsf{rwp}\ e_1 \precsim e_2\ \{\Phi\} \end{array}}{\mathsf{rwp}\ e_1 \precsim e_2\ \{\Phi\}}$$

Ignore for now the highlighted assertions, the rule is also sound without them. The rule states that if we own tapes ι and ι' of type N and M where $N < M$, then for any injective function $f : \mathbb{N}_{\leq N} \to \mathbb{N}_{\leq M}$, we add the value m to the ι' tape and—if it exists—the pre-image $f^{-1}(m)$ to the ι tape. At first glance, this rule may seem arbitrary, but this conditional adding of a sample to one tape is crucial to reasoning about rejection samplers, as we saw in a simplified form in §2.2.

Fragmented couplings can be generalized to approximate reasoning and expectation-preserving composition [Aguirre et al. 2024], now also considering the highlighted assertions. Namely, if we own ε error credits we can distribute them uniformly across the branches that are not in the image of f. That is, if the value is added to both tapes, no error is provided, and if a value is only added to the tape on the right-hand side, we pass the error amplified by a factor of $\frac{M+1}{M-N}$.

Error Amplification. Recall that the standard recursion rule WP-REC only works for recursive programs on the left-hand side of the refinement. To reason about recursive functions on the right-hand side of the refinement in Approxis, one uses *error amplification*, which was first introduced in the Eris logic [Aguirre et al. 2024]. Approxis supports the following induction principle for error amplification.

$$\dfrac{0 < \varepsilon \qquad 1 < k \qquad \forall \varepsilon'.(\lightning(k \cdot \varepsilon') \mathrel{-\!\!*} P) \ast \lightning(\varepsilon') \vdash P}{\lightning(\varepsilon) \vdash P}\ \textsc{err-amp}$$

The rule states that to prove P given some positive error credits $\lightning(\varepsilon)$, it suffices to prove P given some arbitrary amount of error credits $\lightning(\varepsilon')$ and an inductive hypothesis for which we need for pay $\lightning(k \cdot \varepsilon')$ for some $k > 1$. Intuitively, ERR-AMP is sound because given $k > 1$, one can amplify any arbitrary positive error credit by k repeatedly until the error reaches 1, at which point we can derive False by spending 1 error credit. This induction principle encapsulates the kind of repeated amplification we alluded to at the end of §2.2, avoiding the need to manually track how many rounds of amplification are needed.

We specialize the above rule to the following WP-ERR-AMP for reasoning about refinements:

$$\textsc{wp-err-amp}\quad \dfrac{0 < \varepsilon \qquad 1 < k \qquad \forall \varepsilon'.(\lightning(k \cdot \varepsilon') \mathrel{-\!\!*} \mathsf{rwp}\ e \precsim e'\ \{\Phi\}) \ast \lightning(\varepsilon') \vdash \mathsf{rwp}\ e \precsim e'\ \{\Phi\}}{\lightning(\varepsilon) \vdash \mathsf{rwp}\ e \precsim e'\ \{\Phi\}}$$

4.2 Revisiting Rejection Samplers

Now that we have seen the rules of Approxis, we return to the rejection sampler example from §2.2 and describe its proof in more detail. Consider the two programs below, where $M < N$ (for now, it suffices to ignore the lines of code in gray), which reproduce the example from before.

```
let direct _ =                    let reject _ =
  let ι_d = tape M in               let ι_r = tape N in
  rand M ι_d                        (rec sampler _ =
                                       let x = rand N ι_r in
                                       if x ≤ M then x else sampler ()) ()
```

On the left, direct is a simple program that samples directly from rand M. On the right, reject is a rejection sampler. We aim to prove that they compute the same distribution. To do so, we include extra code (in gray) to initialize and use presampling tapes. It is straightforward to use Approxis to prove that the programs without tapes have the same execution distribution as their tape-annotated counterparts (which we omit for brevity).

By applying the *error limiting adequacy result* (Corollary 4.2), it is enough to assume we own an arbitrary and positive amount ε of error credits, and prove the two following assertions:

$$\varepsilon > 0 * ↯(\varepsilon) \vdash \mathsf{rwp}\ \mathsf{direct}\ () \precsim \mathsf{reject}\ ()\ \{v, v'.\, v = v'\}$$
$$\varepsilon > 0 * ↯(\varepsilon) \vdash \mathsf{rwp}\ \mathsf{reject}\ () \precsim \mathsf{direct}\ ()\ \{v, v'.\, v = v'\}\,.$$

We just show the first one, as the second one is mostly analogous (and in fact can be done without error credits by using the wp-rec rule). We begin by applying symbolic execution rules (wp-alloc-tape-l) to allocate the tapes on both sides:

$$\varepsilon > 0 * ↯(\varepsilon) * \iota_d \hookrightarrow (M, \epsilon) * \iota_r \hookrightarrow_{\mathsf{s}} (N, \epsilon) \vdash \mathsf{rwp}\ \mathsf{rand}\, M\ \iota_d \precsim \mathsf{sampler}\ ()\ \{v, v'.\, v = v'\}\,.$$

Although sampler is a recursive function, we cannot apply wp-rec as it appears on the right-hand side. Instead, we leverage the error amplification proof technique and apply wp-err-amp, with amplification factor $k \triangleq \frac{N+1}{N-M}$. Here Φ represents the inductive hypothesis we obtain:

$$\Phi * ↯(\varepsilon') * \iota_d \hookrightarrow (M, \epsilon) * \iota_r \hookrightarrow_{\mathsf{s}} (N, \epsilon) \vdash \mathsf{rwp}\ \mathsf{rand}\, M\ \iota_d \precsim \mathsf{sampler}\ ()\ \{v, v'.\, v = v'\}$$
$$\text{where } \Phi \triangleq ↯(k \cdot \varepsilon') * \iota_d \hookrightarrow (M, \epsilon) * \iota_r \hookrightarrow_{\mathsf{s}} (N, \epsilon) \mathrel{-\!\!*} \mathsf{rwp}\ \mathsf{rand}\, M\ \iota_d \precsim \mathsf{sampler}\ ()\ \{v, v'.\, v = v'\}$$

We now continue by applying wp-fragmented-r-exp, choosing $f \triangleq \lambda x.x$. This consumes our error credit $↯(\varepsilon')$ and distributes it unevenly across the branches depending on the sampling result. We then proceed with a case split. In our first case, both tapes presample the same $v \leq M$:

$$\Phi * \iota_d \hookrightarrow (M, [v]) * \iota_r \hookrightarrow_{\mathsf{s}} (N, [v]) \vdash \mathsf{rwp}\ \mathsf{rand}\, M\ \iota_d \precsim \mathsf{sampler}\ ()\ \{v, v'.\, v = v'\}$$

Then, by taking primitive steps we return the same v on both sides.

In our second case, we only push a value $v > M$ into the right-hand side tape ι and we additionally have $↯(k \cdot \varepsilon')$ error credits:

$$\Phi * ↯(k \cdot \varepsilon') * \iota_d \hookrightarrow (M, \epsilon) * \iota_r \hookrightarrow_{\mathsf{s}} (N, [v]) \vdash \mathsf{rwp}\ \mathsf{rand}\, M\ \iota_d \precsim \mathsf{sampler}\ ()\ \{v, v'.\, v = v'\}$$

We can now take steps only on the right-hand side. The sampling instruction will read v from the tape, and the conditional will evaluate to the else branch, which will leave us to prove:

$$\Phi * ↯(k \cdot \varepsilon') * \iota_d \hookrightarrow (M, \epsilon) * \iota_r \hookrightarrow_{\mathsf{s}} (N, \epsilon) \vdash \mathsf{rwp}\ \mathsf{rand}\, M\ \iota_d \precsim \mathsf{sampler}\ ()\ \{v, v'.\, v = v'\}$$

Notice now we have amplified the error by a factor of exactly k, and hence we can directly apply Φ, our hypothesis we obtained from wp-err-amp to conclude the overall proof.

5 Logical Refinement

It is often hard to reason directly about contextual equivalence, due to the quantification over contexts. As in previous work [Gregersen et al. 2024] we define a *logical refinement relation* to help us reason about contextual refinement. Like contextual refinement, logical refinement is a typed relation: it ranges over pairs of expressions e_1, e_2 and types τ such that e_1 and e_2 have type τ. As we will show later, logical refinement implies contextual refinement. However, logical refinement (as opposed to contextual refinement) is defined in terms of the relational logic, and thus we can reason about it using the inference rules presented in previous sections:

$$\Delta \vDash e_1 \precsim e_2 : \tau \quad \triangleq \quad \forall \varepsilon > 0.\ \lightning(\varepsilon) \mathrel{-\!\!*} \mathsf{rwp}\ e_1 \precsim e_2 \left\{ v_1, v_2.\ \exists \varepsilon' > 0.\ \lightning(\varepsilon') \ast [\![\tau]\!]_\Delta(v_1, v_2) \right\}$$

Here, $[\![\tau]\!]_\Delta$ denotes the semantic interpretation of type τ, and Δ assigns a semantic interpretation to type variables in the context. Intuitively speaking e_1 logically refines e_2 at type τ if we can couple their executions so that they return values related at the semantic interpretation of τ. The key novelty with respect to prior work is the quantification over error credits: we get to assume ownership of a positive amount of error credits in our proof, as long as we ensure that at the end of it we still have a positive amount left. The quantification over error credits allows us to assume ownership of a non-zero amount of error credits whenever reasoning about logical refinement, in particular we can prove the following rule:

$$\frac{\forall \varepsilon > 0.\ \lightning(\varepsilon) \mathrel{-\!\!*} \Delta \vDash e \precsim e' : \tau}{\Delta \vDash e \precsim e' : \tau}\ \textsc{log-get-err}$$

Since LOG-GET-ERR always allows us to obtain a positive amount of error credits, we can internalize a closed rule for error induction for proving logical refinements that assumes no previous ownership of error credits. This can be seen as lifting the WP-ERR-AMP rule to the logical refinement level:

$$\frac{1 < k \qquad \forall \varepsilon.\ (\lightning(k \cdot \varepsilon) \mathrel{-\!\!*} \Delta \vDash e \precsim e' : \tau) \mathrel{-\!\!*} \lightning(\varepsilon) \mathrel{-\!\!*} \Delta \vDash e \precsim e' : \tau}{\Delta \vDash e \precsim e' : \tau}\ \textsc{log-ind-err}$$

The semantic interpretation $[\![\tau]\!]_\Delta(v_1, v_2)$ of a type τ relates values (which do not need themselves to be syntactically well-typed) that behave *as if* they were equivalent values of type τ. This definition is mostly standard, and is defined as usual by induction on τ and in mutual recursion with the refinement relation, see Appendix A of [Gregersen et al. 2023], as well as [Timany et al. 2024] for a general account. Semantic interpretation of a typing context $[\![\Gamma]\!]_\Delta(\gamma_1, \gamma_2)$ relates two substitutions γ_1, γ_2 whenever for all $x \in \mathrm{dom}\ \Gamma$, $[\![\Gamma(x)]\!]_\Delta(\gamma_1(x), \gamma_2(x))$. Logical refinement can then be extended to open terms as usual:

$$\Delta \mid \Gamma \vDash e_1 \precsim e_2 : \tau \triangleq \forall \gamma_1, \gamma_2. [\![\Gamma]\!]_\Delta(\gamma_1, \gamma_2) \mathrel{-\!\!*} \Delta \vDash e_1\gamma_1 \precsim e_2\gamma_2 : \tau$$

We can prove a compatibility result, which intuitively states that all typing rules preserve the relation. For example, in the case of function application we have:

$$\frac{\Delta \vDash f_1 \precsim f_2 : \tau \to \sigma \qquad \Delta \vDash e_1 \precsim e_2 : \tau}{\Delta \vDash f_1\ e_1 \precsim f_2\ e_2 : \sigma}\ \textsc{compat-app}$$

The compatibility results can be combined into the fundamental lemma of the logical relation in the usual way, *i.e.*, by induction on the typing derivation.

Lemma 5.1. *If* $\Gamma \vdash e : \tau$ *then* $\Gamma \vDash e \precsim e : \tau$.

This logical refinement is a strict extension of the one in Clutch [Gregersen et al. 2024]: we can still prove the same refinements (by not using the error credits at all), but we can also prove *new* refinements that were not provable in *loc. cit.*, in particular refinements involving recursive programs on the right. The refinement relation is still sound with respect to contextual equivalence, as stated below:

THEOREM 5.2 (SOUNDNESS). *Let Ξ be a type variable context, and Δ a context assigning a relational interpretation to all type variables in Ξ. If $\Delta \mid \Gamma \vDash e_1 \precsim e_2 : \tau$ then $\Xi \mid \Gamma \vdash e_1 \precsim_{ctx} e_2 : \tau$.*

The proof follows from compatibility of the logical refinement and the adequacy theorem of our relational logic. In particular, we use Corollary 4.2 to erase the quantification over positive errors.

Example 5.3. Logical refinement can be used to prove contextual equivalence of the two samplers considered in §4.2. The statements we have to prove are below:

$$\vDash \mathsf{direct} \precsim \mathsf{reject} : \mathsf{unit} \to \mathsf{nat}, \qquad \vDash \mathsf{reject} \precsim \mathsf{direct} : \mathsf{unit} \to \mathsf{nat}$$

Note that, as opposed to the proof in §4.2, there is no need to assume ownership of a positive amount of error credits, and we can have a closed proof at the level of the logical refinement.

Using similar ideas, Appendix A.1 presents a proof of contextual equivalence between a direct sampler over rand 5 (i.e., a die) and a rejection sampler that simulates it with 3 coin flips, encodes the result as a number from 0 to 7 and returns it if it is 5 or less, retrying otherwise. In particular, this example uses our many-to-one coupling rules (WP-MANY-TO-ONE).

6 Case Studies

In this section we give an overview of several complex examples that we have verified using Approxis. Complete details about each example can be found in the accompanying Coq development.

6.1 The PRP/PRF Switching Lemma, Revisited

In §2.1, we sketched a "weak Switching Lemma", where we considered one particular adversary. We are now ready to prove the "full" version of the switching lemma for arbitrary adversary $\mathcal{A}$. Of course, *some* restrictions on the adversary are still required. First, the adversary must treat the RP/RF as a black box, and not directly access the underlying map that stores the function's values. Second, we must ensure that the RP/RF can only be queried up to Q times. To enforce the first requirement, we require that $\mathcal{A}$ is a well-typed RandML program. For the second requirement, we wrap the RP/RF with the higher-order function q_calls, which uses local state that tracks how many queries have been performed:

```
let q_calls (Q : int) (f : α → β) : α → β option =
  let counter = ref 0 in
  λ x. if (! counter < Q) then incr counter ; Some (f x) else None
```

Our goal is then to prove the following logical refinement (as well as the other direction, the proof of which is similar and omitted) for any $\mathcal{A}$ of type (int → int option) → bool,

$$\text{↯}(\varepsilon) \mathrel{-\!\!*} \vDash \mathcal{A}\ (\mathsf{q\_calls}\ Q\ (\mathsf{irp}\ N)) \precsim \mathcal{A}\ (\mathsf{q\_calls}\ Q\ (\mathsf{irf}\ N)) : \mathsf{bool} \tag{4}$$

where $\varepsilon = \frac{Q(Q-1)}{2N}$. By unfolding the definition of the logical relation and applying the adequacy theorem of Approxis to (4), these refinements imply Lemma 2.1.

In order to reason about the unknown program $\mathcal{A}$, we leverage the logical relation. Specifically, from the assumption that $\mathcal{A}$ is well typed together with the Fundamental Lemma (Lemma 5.1), we derive an Approxis specification for $\mathcal{A}$ of the form $\vDash \mathcal{A} \precsim \mathcal{A} : ((\mathsf{int} \to \mathsf{int\ option}) \to \mathsf{bool})$. Using this and the logical relation's compatibility rule for function application leaves us to prove:

$$\text{↯}(\varepsilon) \mathrel{-\!\!*} \ \vDash (\mathsf{q\_calls}\ Q\ (\mathsf{irp}\ N)) \precsim (\mathsf{q\_calls}\ Q\ (\mathsf{irf}\ N)) : \mathsf{int} \to \mathsf{int\ option} \tag{5}$$

In other words, after using the logical relation, the goal that remains makes no reference to the unknown code for the adversary, $\mathcal{A}$. From here on, the proof is very similar to that of the weak PRP/PRF Switching Lemma. We symbolically evaluate both programs, which results in the allocation of the list of unused values in irp, the finite maps that both irp and irf use, and a counter for each program. Let rp_Q denote the function returned by q_calls Q (irp N), and likewise rf_Q for irf. The key difference with the proof in §2.1 is that, instead of proving the ε-equivalence of two for loops by induction on Q, we prove the ε-equivalence of two *functions* by establishing an invariant that holds before and after all calls to rp_Q and rf_Q. This invariant we need states that both counters point to the same value i, that the maps m and m′ are equal, and that the current error budget is $\varepsilon_i = \sum_{k=i}^{Q-1} \frac{k}{N}$ when the counter is $0 \leq i \leq Q$. Furthermore, the list of unused values has length (at least) $N - i$. Since $\lightning(\varepsilon_i)$ is an ordinary proposition in Approxis, and since Approxis is an impredicative higher-order logic, we can simply "store" error credits in the invariant.

The semantic interpretation of function types requires us to prove that both functions in (5) map related arguments to related results. In other words, we can show the refinement by applying both functions to the same integer n. Just as in §2.1, we can then argue that with probability at most[2] $\frac{N-(N-i)}{N} = \frac{i}{N}$ a collision occurs. By WP-COUPLE-RAND-RAND-ERR-LE, we can force both functions to sample the same value by paying $\lightning\left(\frac{i}{N}\right)$. This is exactly the first element in the sum ε_i. The functions thus return the same results, as required. It remains to re-establish the function invariant. Since the counters have been incremented to $i + 1$, we only need to give back ε_{i+1} credits, which is exactly what is left of $\lightning(\varepsilon_i)$ after splitting off $\lightning\left(\frac{i}{N}\right)$.

In conclusion, thanks to the logical relation, proving the full Switching Lemma is as simple as proving the weak version.

6.2 IND$-CPA Security of Symmetric Encryption

A key notion of security for a symmetric (*i.e.*, private key) encryption scheme is "indistinguishability from random under chosen-plaintext attacks" (IND$-CPA, a.k.a. CPA$). The IND$-CPA "advantage" of an adversary $\mathcal{A}$ against an encryption scheme (keygen, enc, dec) is defined as the probability that $\mathcal{A}$ is able to distinguish a ciphertext c corresponding to a plain-text message msg from a randomly chosen ciphertext c′, even if $\mathcal{A}$ can choose msg (see, *e.g.*, [Rosulek 2021, Def. 7.2]). For an adversary that can make only up to Q-queries to the encryption oracle, the advantage is thus equal to

$$|\Pr[\mathcal{A}(\text{q\_calls } Q \text{ (enc (keygen ())))}] - \Pr[\mathcal{A}(\text{q\_calls } Q \text{ (rand\_cipher))}]| \tag{6}$$

where rand_cipher = λ msg. (rand N, rand N) produces random ciphertexts.

It is well-known that a *deterministic* encryption scheme cannot achieve IND$-CPA security [Katz and Lindell 2021; Rosulek 2021]. A standard solution to obtain a IND$-CPA secure scheme is to randomize the encryption function. We exemplify this idea by proving a bound on the IND$-CPA advantage for the following textbook construction [Rosulek 2021, Def. 7.4][Katz and Lindell 2021, Def. 3.28] of a symmetric scheme from a random function:

```
let enc prf key msg =  let r = rand N in        let keygen ()         = rand N
                       let pad = prf key r in   let dec prf key (r, c) = let pad = prf key r in
                       let c = xor msg pad in                            let msg = xor c pad in
                       (r, c)                                            msg
```

In particular, we prove the following refinement (and its converse) in Approxis,

$$\lightning(Q^2/(2N)) \twoheadrightarrow\!\!* \vDash \mathcal{A} \text{ (q\_calls } Q \text{ enc}_{rf}) \precsim \mathcal{A} \text{ (q\_calls } Q \text{ rand\_cipher)} : \text{bool} \tag{7}$$

[2] This "worst case" occurs if all previous calls to irp were made with different arguments and List.length l_unused = $N - i$.

where $\mathsf{enc}_{rf} = \mathsf{enc}\ (\mathsf{let}\ rf = \mathsf{irf}\, N\ \mathsf{in}\ \lambda\, \mathsf{key}.\ rf)$. Intuitively, the scheme is secure because *prf* produces random-looking outputs. So long as pad = *prf* key r never repeats throughout the Q calls to enc_{rf}, the xor msg pad acts as a one-time pad, and the ciphertexts look random.

The proof of (7) thus hinges on the fact that the randomly sampled value r never repeats, since this ensures that irf samples a new value for pad. Formally, we argue that after initialization of irf and q_calls, the encryption oracle enc_{rf} and the random cipher oracle satisfy the following invariant, which ties the amount of error credits left to the counter value i.

$$\exists i, m.\ ↯\big((Q^2 - i^2)/(2N)\big) \ast \mathsf{counter} \mapsto i \ast \mathsf{counter}' \mapsto_{\mathsf{s}} i \ast \mathit{is\_rf}\, m \ast |\mathsf{dom}\ m| = i \tag{8}$$

We use the approximate coupling rule to “pay off” the risk of a repeated use of r at the i-th oracle call. The exact source of errors is different here from the Switching Lemma, since we have to argue that the randomly sampled arguments to *rf* do not repeat, whereas in the Switching Lemma, we are concerned with collisions that get sampled if fresh arguments are fed to *rf* and *rp*. The exact rule that allows us to do this is given below, where the list l is instantiated with the domain of the map m which is tracked by *rf* in the invariant (8).

$$\frac{↯\left(\frac{\mathsf{length}(l)}{N+1}\right) \qquad \forall n \le N.\ n \notin l \twoheadrightarrow \mathsf{rwp}\ n \precsim n\ \{\Phi\}}{\mathsf{rwp}\ \ \mathsf{rand}\, N \precsim \mathsf{rand}\, N\ \{\Phi\}}$$

Since the invariant is preserved throughout, the output of enc_{rf} is thus $\frac{Q^2}{2N}$-equivalent to that of rand_cipher in Q oracle calls.

6.3 Sampling from B+ Trees

In this case study, we show the correctness of a rejection sampling scheme developed by Olken and Rotem [1989] for drawing a random sample from a B+ tree. This case study demonstrates how Approxis is able to handle complex mutable state and establish equivalences that rely on type abstraction. A B+ tree [Bayer and McCreight 1972] is a height-balanced tree data structure that is widely used for storing data in filesystems and databases. Unlike a binary search tree, a B+ tree's internal nodes may have more than 2 children, up to some maximum M.

If a B+ tree's nodes include additional *ranking* information recording how many leaves are descendants of each node, then it is straight-forward to draw a random element. For a tree with N total elements, draw a random number uniformly from $\{0, \ldots, N-1\}$ and then use the ranks to find the i-th element in the tree. However, maintaining the ranks has overhead.

Olken and Rotem [1989] developed a rejection sampling algorithm for sampling from a *non*-ranked B+ tree. Starting from the root, the algorithm recursively descends down the tree. At each non-leaf node, it samples a random number i uniformly from $\{0, 1, \ldots, M-1\}$. If the node has an i-th child, the algorithm recurses on it. If the node does not have an i-th child, it aborts early by returning to the root and restarting. Once the algorithm reaches a leaf, it returns it as the selected sample.

We have implemented the sampling algorithm for ranked B+ trees and the rejection sampler for non-ranked trees as two functions called naive_sample and optimized_sample, respectively (see Appendix A.2 for the full code). Our main result for this case study shows that these two functions are equivalent. Of course, they are only equivalent when they operate over well-formed trees, so we state this result as a contextual equivalence about two different implementations of an abstract tree data type. To do so, we first define the following additional functions: init_tree, which takes an integer and returns a B+ tree containing that integer, insert_tree, which inserts an integer into a tree, and build_ranked, which takes a (non-ranked) B+ tree and returns a ranked B+ tree with the same entries and shape. Then the following two packed tuples bundle the B+ tree operations:

$$\mathsf{opt} \triangleq \mathsf{pack}\ (\mathsf{init\_tree}, \mathsf{insert\_tree}, \mathsf{optimized\_sample})$$
$$\mathsf{naive} \triangleq \mathsf{pack}\ (\mathsf{init\_tree}, \mathsf{insert\_tree}, \lambda t. \mathsf{naive\_sample}\ (\mathsf{build\_ranked}\ t))$$

where the sampling routine in naive takes a tree t, builds the ranked version of the tree, and then uses the naïve routine.[3] With these preliminaries in place, our main result can be stated as $\vdash \mathsf{opt} \simeq_{\mathsf{ctx}} \mathsf{naive} : \tau$, where $\tau \triangleq \exists \tau.\ (\mathsf{Int} \to \tau) \times (\tau \times \mathsf{Int} \to 1) \times (\tau \to \mathsf{Int})$.

This proof is described in more detail in Appendix A.2. At a high level, it has two components. First, there is non-probabilistic reasoning showing that the various routines traverse and modify the trees correctly. This makes up the bulk of the proof and consists of traditional separation-logic style reasoning about trees. For this part, Approxis's support for the rich reasoning principles developed in earlier separation logics is essential. The second component is the actual probabilistic reasoning using couplings. Here, the coupling reasoning in this proof is quite similar to the arguments we have already seen using fragmented and many-to-one couplings in simpler rejection samplers.

7 Semantic Model and Soundness

The soundness of Approxis is justified by defining a semantic model of $\mathsf{rwp}\ e \precsim e'\ \{\Phi\}$ in the Iris base logic [Jung et al. 2018]. The base logic is a higher-order separation logic that lacks any connectives for reasoning about programs. In this section, we define the semantic model of rwp and discuss how it implies the existence of an approximate coupling of the execution of the two programs.

The model of rwp is inspired by the model of the (non-approximate) coupling logic Clutch [Gregersen et al. 2024]. We emphasize the following technical novelties and improvements over Clutch's model:

(1) The approach is generalized to approximate couplings and expectation-preserving composition by incorporating error credits [Aguirre et al. 2024] in the relational setting.
(2) The model allows for coupling rules and right-hand side rules to be applied when the left-hand side is a value (a limitation of the general structure of the Clutch model).
(3) The model introduces two new *coupling precondition* connectives and a notion of *erasability* that captures the essence of why asynchronous couplings [Gregersen et al. 2024] are sound.

7.1 Model

The semantic model of rwp is defined using two unary connectives: a weakest precondition $\mathsf{wp}\ e\ \{\Phi\}$ and a separation-logic resource $\mathsf{spec}(e')$ that tracks the right-hand-side specification program, as in prior work on refinement reasoning in separation logic [Gregersen et al. 2024; Timany et al. 2024; Turon et al. 2013]. The rwp is defined as

$$\mathsf{rwp}\ e \precsim e'\ \{\Phi\} \triangleq \forall K.\ \mathsf{spec}(K[e']) \mathrel{-\!\!*} \mathsf{wp}\ e\ \{v.\ \exists v'.\ \mathsf{spec}(K[v']) * \Phi(v, v')\}.$$

By quantifying over evaluation contexts K, we close the definition under evaluation contexts on the right-hand side; for the left-hand side this is not needed as the weakest precondition already satisfies the bind rule. The main challenge of defining the relational connective is thus to define the model of the unary weakest precondition in a suitable way.

In isolation, the weakest precondition $\mathsf{wp}\ e\ \{\Phi\}$ encodes partial correctness: intuitively it means that the execution of e is *safe* (*i.e.*, the probability of crashing is zero) and for every possible return value v, the postcondition $\Phi(v)$ holds. Internally, however, in order to do approximate (relational) reasoning, the weakest precondition pairs up of the probability distribution of individual steps of

[3] Building a ranked tree each time is inefficient, but naive serves as a specification for opt, so its efficiency is not relevant.

the program with the probability distribution of individual steps of *some* other program, in such a way that there exists an approximate coupling among them. Through separation-logic machinery, we tie this "other" program to the program tracked by the $\mathsf{spec}(e')$ resource, and the approximation error to error credits $\text{↯}(\varepsilon)$. The weakest precondition itself satisfies all the usual program logic rules that one would expect and we refer to Figure 5 for an overview.

Weakest Precondition. The definition of the weakest precondition is shown below. As done throughout this paper, we ignore the general connectives that are used for manipulating Iris-style ghost resources and invariants, *i.e.*, the *update modality* $\mid\!\Rrightarrow$ and *invariant masks* ($\mathcal{E}$, $\emptyset$) as found in Iris [Jung et al. 2018], which are orthogonal to the core challenges that we address. Our use is standard and the weakest precondition can be understood by omitting the grayed out parts. The definition looks as follows.

$$
\begin{aligned}
\mathsf{wp}_{\mathcal{E}}\ e_1\ \{\Phi\} \triangleq{}& \forall \sigma_1, \rho'_1, \varepsilon_1.\ S(\sigma_1, \rho'_1, \varepsilon_1) \mathrel{-\!\!*} \\
&\quad {}_{\mathcal{E}}\!\mid\!\Rrightarrow_{\emptyset} \mathsf{scpl}\ \sigma_1 \precsim_{\varepsilon_1} \rho'_1\ \{\sigma_2, \rho'_2, \varepsilon_2. \\
&\qquad \big(e_1 \in \mathit{Val} * {}_{\emptyset}\!\mid\!\Rrightarrow_{\mathcal{E}} S(\sigma_2, \rho'_2, \varepsilon_2) * \Phi(e_1)\big) \lor \\
&\qquad \big(e_1 \notin \mathit{Val} * \mathsf{pcpl}\ (e_1, \sigma_2) \precsim_{\varepsilon_2} \rho'_2\ \{e_2, \sigma_3, \rho'_3, \varepsilon_3. \\
&\qquad\quad \triangleright\, \mathsf{scpl}\ \sigma_3 \precsim_{\varepsilon_3} \rho'_3\ \{\sigma_4, \rho'_4, \varepsilon_4.\ {}_{\emptyset}\!\mid\!\Rrightarrow_{\mathcal{E}} S(\sigma_4, \rho'_4, \varepsilon_4) * \mathsf{wp}_{\mathcal{E}}\ e_2\ \{\Phi\}\}\}\big)\}
\end{aligned}
$$

The connective is defined as a *guarded fixed point* of the equation above, as is custom in many program logics for partial correctness. The fixed point exists because the recursive occurrence of the weakest precondition occurs under the later modality $\triangleright$ [Jung et al. 2016].

The diagram below shows how the two programs are updated in a single unrolling of the weakest precondition. This will contain a single execution step on the left-hand side, preceded and followed by a sequence of execution steps on the right-hand side. Along these steps, the tapes can be updated with new samples, with errors threaded through to construct approximate couplings between steps:

$$\begin{tikzcd}
(e_1, \sigma_1) \sim_{\varepsilon_1} \rho'_1 \arrow[r, "{\substack{\text{RHS exec}_n,\\ \text{update tapes}}}", "\mathsf{scpl}"'] & (e_1, \sigma_2) \sim_{\varepsilon_2} \rho'_2 \arrow[r, "{\substack{\text{LHS exec}_1,\ \text{RHS exec}_n,\\ \text{update tapes}}}", "\mathsf{pcpl}"'] & (e_2, \sigma_3) \sim_{\varepsilon_3} \rho'_3 \arrow[r, "{\substack{\text{RHS exec}_n,\\ \text{update tapes}}}", "\mathsf{scpl}"'] & (e_2, \sigma_4) \sim_{\varepsilon_4} \rho'_4
\end{tikzcd}$$

We now explain the definition in detail. We first start by assuming ownership of a *state interpretation* $S(\sigma_1, \rho'_1, \varepsilon_1)$. The state interpretation predicate $S : \mathit{State} \to \mathit{Cfg} \to [0,1] \to \mathit{iProp}$ interprets the physical state of the program, the specification program, and the approximation error as resources in Approxis which, *e.g.*, gives meaning to the points-to connective $\ell \mapsto v$, the specification resource $\mathsf{spec}(e')$, and error credits $\text{↯}(\varepsilon)$. Our choice of resource algebras is standard (see Gregersen et al. [2024] and Aguirre et al. [2024] for more details) and it is sufficient to know that they reflect the (partial) knowledge that the logical connectives represent. For instance, for the heap points-to connective we have that $S(\sigma_1, \rho'_1, \varepsilon_1) * \ell \mapsto v \vdash \sigma_1(\ell) = v$; for the specification resource we have $S(\sigma_1, (e'_1, \sigma'_1), \varepsilon_1) * \mathsf{spec}(e') \vdash e'_1 = e'$; and for error credits we have $S(\sigma_1, \rho'_1, \varepsilon_1) * \text{↯}(\varepsilon) \vdash \varepsilon_1 \geq \varepsilon$.

Second, we have to prove a *spec-coupling precondition* $\mathsf{scpl}\ \sigma_1 \precsim_{\varepsilon_1} \rho'_1\ \{\ldots\}$. We define the connective in Figure 4, but in essence it allows the *right-hand-side* program to be progressed. Intuitively, $\mathsf{scpl}\ \sigma_1 \precsim_{\varepsilon_1} \rho'_1\ \{\sigma_2, \rho'_2, \varepsilon_2. P\}$ says that with error budget ε_1 there exists a (possibly empty) sequence of composable approximate couplings starting from state σ_1 and configuration ρ'_1 that ends up in some state σ_2 and configuration ρ'_2 with leftover error budget ε_2, such that the proposition P holds. By allowing the left-hand-side *state* to be progressed with the right-hand-side *configuration*, we permit certain *asynchronous* coupling rules as discussed below in detail.

$$\textsc{spec-coupl-err-1}\quad \frac{}{\mathsf{scpl}\ \sigma \precsim_{1} \rho\ \{\Phi\}} \qquad\qquad \textsc{spec-coupl-ret}\quad \frac{\Phi(\sigma, \rho', \varepsilon)}{\mathsf{scpl}\ \sigma \precsim_{\varepsilon} \rho'\ \{\Phi\}}$$

$$\textsc{spec-coupl-exp}\quad \frac{\begin{array}{c} \mu_1 \precsim_{\varepsilon_1} (\mu_1' \gg\!= \lambda\, \sigma_2'.\, \mathsf{step}_n(e_1', \sigma_2')) : R \qquad \varepsilon_1 + \mathbb{E}[\mathcal{E}_2] \leq \varepsilon \\ \mathsf{erasable}(\mu_1, \sigma_1) \qquad \mathsf{erasable}(\mu_1', \sigma_1') \qquad \forall (\sigma_2, \rho_2') \in R.\ {\mid\!\Rrightarrow_{\emptyset}}\, \mathsf{scpl}\ \sigma_2 \precsim_{\mathcal{E}_2(\rho_2')} \rho_2'\ \{\Phi\} \end{array}}{\mathsf{scpl}\ \sigma_1 \precsim_{\varepsilon} (e_1', \sigma_1')\ \{\Phi\}}$$

Fig. 4. Inductive definition of the specification-coupling precondition $\mathsf{scpl}\ \sigma \precsim_{\varepsilon} \rho\ \{\Phi\}$.

Next, if e_1 is a value we have to return the updated state interpretation and prove the postcondition $\Phi(e_1)$. If e_1 is *not* a value, we have to prove a *program-coupling precondition* $\mathsf{pcpl}\ (e_1, \sigma_2) \precsim_{\varepsilon_2} \rho_2'\ \{\ldots\}$. We define the connective formally below, but it allows the *left-hand-side* program to take a single step and the *right-hand-side* program to take a finite number of steps. Intuitively, $\mathsf{pcpl}\ \rho_1 \precsim_{\varepsilon_1} \rho_1'\ \{\rho_2, \rho_2', \varepsilon_2.\, P\}$ says that with error budget ε_1 there exists an approximate coupling of a *single* step of configuration ρ_1 with a finite number of steps of configuration ρ_1' that ends up in configurations ρ_2 and ρ_2' with leftover error budget ε_2, such that the proposition P holds.

Finally, under a later modality (which signifies that a step of e_1 has been taken), we have to prove another spec-coupling precondition before returning the updated state interpretation and showing that $\mathsf{wp}\ e_2\ \{\Phi\}$ holds recursively. The second occurrence of the spec-coupling precondition can mostly be ignored and is only required to validate the invariant opening rule which we omit.

Coupling Preconditions. The spec-coupling precondition is defined inductively by the inference rules shown in Figure 4. If the error budget is 1 or if the postcondition holds for the input parameters, the spec-coupling precondition holds trivially (SPEC-COUPL-ERR-1 and SPEC-COUPL-RET, respectively). The last constructor (SPEC-COUPL-EXP) is by far the most interesting: it allows us to incorporate approximate couplings and requires the existence of an (ε_1, R)-coupling of μ_1 and $(\mu_1' \gg\!= \lambda\, \sigma_2'.\, \mathsf{step}_n(e_1', \sigma_2'))$ for prover-chosen μ_1, μ_1' and n. Here $\mathsf{step}_n : \mathit{Cfg} \to \mathcal{D}(\mathit{Cfg})$ denotes n steps of partial execution, *i.e.*,

$$\mathsf{step}_n(e, \sigma) \triangleq \begin{cases} \mathsf{ret}(e, \sigma) & \text{if } e \in \mathit{Val} \ \text{ or } \ n = 0, \\ \mathsf{step}(e, \sigma) \gg\!= \mathsf{step}_{(n-1)} & \text{otherwise.} \end{cases}$$

and $\mu_1, \mu_1' \in \mathcal{D}(\mathit{State})$ are arbitrary *erasable* distributions (*c.f.*, Definition 3.12) w.r.t. σ_1 and σ_1'.

Finally, to allow expectation-preserving composition (*c.f.*, Lemma 3.10), the prover picks an error function $\mathcal{E}_2$ such that $\varepsilon_1 + \mathbb{E}[\mathcal{E}_2] \leq \varepsilon$, where the expectation $\mathbb{E}[\mathcal{E}_2]$ is computed with respect to the distribution $(\mu_1' \gg\!= \lambda\, \sigma_2'.\, \mathsf{step}_n(e_1', \sigma_2'))$. Then for all σ_2 and ρ_2' in the support of the coupling, the spec-coupling precondition must hold recursively with the new error budget $\mathcal{E}_2(\rho_2')$.

The program-coupling precondition is defined in a similar style to the SPEC-COUPL-EXP constructor, but the approximate coupling requires *exactly* one step on the left-hand side as seen below.

$$\frac{\begin{array}{c} \mathsf{step}(e_1, \sigma_1) \precsim_{\varepsilon_1} (\mu_1' \gg\!= \lambda\, \sigma_2'.\, \mathsf{step}_n(e_1', \sigma_2')) : R \qquad \mathsf{red}(e_1, \sigma_1) \\ \varepsilon_1 + \mathbb{E}[\mathcal{E}_2] \leq \varepsilon \qquad \mathsf{erasable}(\mu_1', \sigma_1') \qquad \forall ((e_2, \sigma_2), (e_2', \sigma_2')) \in R.\ {\mid\!\Rrightarrow_{\emptyset}}\, \Phi(e_2, \sigma_2, e_2', \sigma_2', \mathcal{E}_2(e_2, \sigma_2)) \end{array}}{\mathsf{pcpl}\ (e_1, \sigma_1) \precsim_{\varepsilon} (e_1', \sigma_1')\ \{\Phi\}}$$

The left-hand side program is also required to be reducible (to guarantee safety), and for all configurations in the support of the coupling, the postcondition must hold. Note that the expectation $\mathbb{E}[\mathcal{E}_2]$ is taken with respect to the distribution $\mathsf{step}(e_1, \sigma_1)$. This guarantees that every recursive unfolding of the weakest precondition corresponds to a single step of the left-hand-side program which is essential to validating the standard program logic rules found in Figure 5.

$$
\begin{aligned}
e_1 \overset{\text{pure}}{\rightsquigarrow} e_2 * \mathsf{wp}\ e_2\ \{\Phi\} &\vdash \mathsf{wp}\ e_1\ \{\Phi\} \\
\forall \ell.\, \ell \mapsto v \mathrel{-\!\!*} \Phi(\ell) &\vdash \mathsf{wp}\ \mathsf{ref}\, v\ \{\Phi\} \\
(\ell \mapsto v \mathrel{-\!\!*} \Phi(v)) * \ell \mapsto v &\vdash \mathsf{wp}\ !\,\ell\ \{\Phi\} \\
(\ell \mapsto w \mathrel{-\!\!*} \Phi(())) * \ell \mapsto v &\vdash \mathsf{wp}\ \ell \leftarrow w\ \{\Phi\} \\
\forall n \leq N.\, \Phi(n) &\vdash \mathsf{wp}\ \mathsf{rand}\, N\ \{\Phi\} \\
\Phi(v) &\vdash \mathsf{wp}\ v\ \{\Phi\} \\
\mathsf{wp}\ e\ \big\{v.\, \mathsf{wp}\ K[v]\ \{\Phi\}\big\} &\vdash \mathsf{wp}\ K[e]\ \{\Phi\} \\
(\forall v.\, \Psi(v) \mathrel{-\!\!*} \Phi(v)) * \mathsf{wp}\ e\ \{\Psi\} &\vdash \mathsf{wp}\ e\ \{\Phi\} \\
P * \mathsf{wp}\ e\ \{\Phi\} &\vdash \mathsf{wp}\ e\ \{v.\, P * \Phi(v)\} \\
\forall \iota.\, \iota \hookrightarrow (N, \epsilon) \mathrel{-\!\!*} \Phi(\iota) &\vdash \mathsf{wp}\ \mathsf{tape}\, N\ \{\Phi\} \\
(\forall n \leq N.\, \iota \hookrightarrow (N, \epsilon) \mathrel{-\!\!*} \Phi(n)) * \iota \hookrightarrow (N, \epsilon) &\vdash \mathsf{wp}\ \mathsf{rand}\, N\ \iota\ \{\Phi\} \\
(\iota \hookrightarrow (N, \vec{n}) \mathrel{-\!\!*} \Phi(n)) * \iota \hookrightarrow (N, n \cdot \vec{n}) &\vdash \mathsf{wp}\ \mathsf{rand}\, N\ \iota\ \{\Phi\}
\end{aligned}
$$

Fig. 5. Standard weakest-precondition rules.

The lemmas below illustrate how spec-coupling and program-coupling preconditions interact with the operational semantics to allow us to construct couplings for program executions. First, we see the case of the spec-coupling precondition:

Lemma 7.1. *Let (e, σ_1) and ρ'_1 be configurations for the left-hand-side and right-hand-side programs, and let $\varphi \subseteq Val \times Val$ be a relation on values. If, for some error $\varepsilon_1 \in [0, 1]$,*

$$\mathsf{scpl}\ \sigma_1 \precsim_{\varepsilon_1} \rho'_1\ \left\{\sigma_2, \rho'_2, \varepsilon_2.\, \mathsf{exec}_m(e, \sigma_2) \precsim_{\varepsilon_2} \mathsf{exec}(\rho'_2) : \varphi\right\},$$

then there exists a (ε_1, φ)-coupling $\mathsf{exec}_m(e, \sigma_1) \precsim_{\varepsilon_1} \mathsf{exec}(\rho'_1) : \varphi$.

The program-coupling precondition satisfies an analogous result, but notice the extra computation step in the conclusion:

Lemma 7.2. *Let (e_1, σ_1) and ρ'_1 be configurations for the left-hand-side and right-hand-side programs where $e_1 \notin Val$, and let $\varphi \subseteq Val \times Val$ be a relation on values. If, for some error ε_1,*

$$\mathsf{pcpl}\ (e_1, \sigma_1) \precsim_{\varepsilon_1} \rho'_1\ \left\{e_2, \sigma_2, \rho'_2, \varepsilon_2.\, \mathsf{exec}_m(e_2, \sigma_2) \precsim_{\varepsilon_2} \mathsf{exec}(\rho'_2) : \varphi\right\},$$

then there exists a (ε_1, φ)-coupling $\mathsf{exec}_{m+1}(e_1, \sigma_1) \precsim_{\varepsilon_1} \mathsf{exec}(\rho'_1) : \varphi$.

The proofs of these auxiliary lemmas rely on erasability as well as Lemmas 3.9 and 3.10 to construct the coupling of the executions.

7.2 Soundness

Soundness of Approxis and the relational program logic follows from the adequacy theorem below.

Theorem 7.3 (Adequacy). *Let $\varphi \subseteq Val \times Val$ be a relation over values and let $0 \leq \varepsilon \leq 1$. If $\mathsf{spec}(e') * \text{↯}(\varepsilon) \vdash \mathsf{wp}\ e\ \{v.\, \exists v'.\, \mathsf{spec}(v') * \varphi(v, v')\}$ then $\forall \sigma, \sigma'.\, \mathsf{exec}(e, \sigma) \precsim_{\varepsilon} \mathsf{exec}(e', \sigma') : \varphi$.*

The proof has a similar structure to the soundness theorem of Clutch [Gregersen et al. 2024]. By continuity, it suffices to show the following approximate coupling:

$$\mathsf{exec}_n(e, \sigma) \precsim_{\varepsilon} \mathsf{exec}(e', \sigma') : \varphi$$

for all σ, σ', and n. The theorem then follows by induction on n. The interesting case is the inductive step, when $n = m + 1$. After unfolding the definition of the weakest precondition, we can apply Lemma 7.1 and Lemma 7.2 to construct a coupling between $\mathsf{exec}_{m+1}(e, \sigma)$ and $\mathsf{exec}(e', \sigma')$.

8 Related Work

Probabilistic Couplings. Relational reasoning about program via (exact) probabilistic couplings can be traced back to pRHL [Barthe et al. 2015, 2009] and was later extended to support approximate couplings in apRHL [Barthe et al. 2016a, 2012], apRHL+ [Barthe et al. 2016c], and EpRHL [Barthe et al. 2017]. These approximate logics can be used to reason about a wide range of properties such as differential privacy and expected sensitivity, but they are limited to reasoning about first-order programs. Aguirre et al. [2021] introduce HO-RHL, which use couplings to reason about adversarial computations in a higher-order setting. HO-RHL, however, only allows synchronous couplings and only supports first-order global state and structural recursion. Clutch [Gregersen et al. 2024] introduces asynchronous couplings in a higher-order setting with higher-order local state. However, Clutch does not support approximate or fragmented couplings.

Approximate Reasoning. Aside from relational approaches, approximate reasoning has also been used in the unary setting. The unary logic aHL [Barthe et al. 2016b] is used to reason about accuracy properties of first-order randomized algorithms, where errors are tracked by a grading on Hoare triples. Eris [Aguirre et al. 2024] extends this to the higher-order setting and tracks error probability as a separation logic resource. In a slightly different line of work, expectation-based logics [Batz et al. 2019, 2023; Kaminski et al. 2016; Morgan et al. 1996] can also be used to reason about approximate correctness of first-order imperative probabilistic programs. In particular, eRHL [Avanzini et al. 2024] supports reasoning about asynchronous samplings via ⋆-couplings.

Resource Reasoning with Credits. Using sub-structural credits to track a program's resource consumption was pioneered in type systems for automated amortized resource analysis (AARA) [Hofmann and Jost 2003]. Subsequent research extends this approach to reason about expected cost bounds in probabilistic programs [Das et al. 2023; Ngo et al. 2018; Wang et al. 2020]. Inspired by AARA, Atkey [2011] introduced resource-tracking credits in separation logic to reason about amortized resource consumption. A variant of this idea is implemented as time credits in Iris to reason about running time complexity of higher-order programs [Charguéraud and Pottier 2019; Mével et al. 2019; Pottier et al. 2024] and expected running time in Tachis [Haselwarter et al. 2024]. Eris [Aguirre et al. 2024] uses error credits to reason about error bounds of higher-order probabilistic programs, which Approxis adapts to the relational setting.

Logical Relations and Probability. Bizjak and Birkedal [2015] developed a logical relations model of a higher-order probabilistic programming languages involving both state and discrete probabilistic choice to reason contextual equivalence. The approach was then extended to support continuous probabilistic choice [Culpepper and Cobb 2017; Wand et al. 2018], recursively nested queries [Zhang and Amin 2022], and non-determinism [Aguirre and Birkedal 2023]. The logical relation developed in Clutch [Gregersen et al. 2024] supports asynchronous couplings and is very similar to the model in our work. Our logical relation, however, also supports proving contextual equivalences by means of approximation. This is key to proving equivalences of rejection sampling programs which, to our knowledge, is out of scope for previous models based on logical relations.

Besides contextual equivalence, logical relations are used to reason about contextual distance between probabilistic programs [Crubillé and Dal Lago 2017; Crubillé and Dal Lago 2015]. Contextual distance can be seen as a generalization of contextual equivalence into a metric for analyzing distances between probabilistic programs. Using error credits to reason about contextual distances is an interesting avenue for future work.

Separation Logic and Probability. In addition to Eris [Aguirre et al. 2024] and Clutch [Gregersen et al. 2024], more tangentially to our work, Batz et al. [2019] developed a weakest precondition calculus for quantitative reasoning about probabilistic pointer programs in QSL, a quantitative analog of classical separation logic. A different line of work develop separation logics in which separating conjunction models probabilistic independence. This was first explored in probabilistic separation logic (PSL) [Barthe et al. 2019] and subsequently extended to reason about conditional independence [Bao et al. 2021, 2024; Li et al. 2023] and negative dependence [Bao et al. 2022].

Program Logics for Cryptographic Security. CertiCrypt [Barthe et al. 2009, 2010] is a framework implemented in Coq [Team 2024] for verifying code-based cryptographic proofs. Programs in CertiCrypt are written in pWhile, a probabilistic imperative language, and the logic is based on pRHL [Barthe et al. 2009]. CertiCrypt can prove approximate results such as the PRP/PRF lemma only at the level of the Coq meta-logic, since the program logic itself is based on exact couplings.

Building on pRHL and CertiCrypt, EasyCrypt [Barthe et al. 2014] is a stand-alone tool for cryptography, integrating automation via SMT solvers. Although EasyCrypt can reason about simple rejection samplers [Almeida et al. 2023], existing proofs require analysing the probability of each outcome, instead of a relational equivalence proof. Rejection samplers with dynamic references or sophisticated early-abort, like the B+ tree sampler, would be difficult to do in this setting.

SSProve [Abate et al. 2021; Haselwarter et al. 2023] is a framework implemented in Coq [Team 2024] for writing so-called state-separating proofs [Brzuska et al. 2018]. Based on a monadic pRHL-like logic [Maillard et al. 2020], games in SSProve are split into packages operating on disjoint states, which enables some amount of modular reasoning. In contrast to the language considered by Approxis, SSProve is first order and does not support dynamically allocated local state.

9 Conclusion

We presented Approxis, the first higher-order separation logic for approximate relational reasoning. In addition to establishing approximate bounds between probabilistic programs, we developed a novel logical relation in Approxis for proving contextual refinement, by parameterizing over arbitrary positive error. We demonstrated the strengths of Approxis on various case studies involving higher-order, local state, and non-trivial rejection sampling behavior. Using Approxis, we proved both approximate and exact examples, and also used the logical relation to establish contextual refinements of examples that were previously out of scope.

We believe Approxis opens up numerous avenues for future work related to security of cryptographic protocols. Firstly, we would like to extend Approxis to reason about concurrent programs in order to prove security guarantees of distributed systems. Secondly, it would be interesting to explore how Approxis can be further improved to reason about time complexity of programs, to bound the computational power of adversaries. Finally, we aim to modify Approxis to reason about several other security properties, including differential privacy and probabilistic sensitivity.

Acknowledgments

This work was supported in part by the National Science Foundation, grant no. 2338317, the Carlsberg Foundation, grant no. CF23-0791, a Villum Investigator grant, no. 25804, Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation, and the European Union (ERC, CHORDS, 101096090). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

A Extended Case Studies

A.1 Simulating Dice

Rejection samplers are a kind of Las Vegas algorithms used to simulate complicated probability distributions with simple probabilistic primitives. These algorithms loop repeatedly, only terminating when it produces an acceptable value, one that correspond to a value in the target distribution. Previously in §4.2, we showed how to prove a simple rejection sampler is equivalent to a rand expression.

To show that our techniques scale, we show a slightly more complicated implementation of a rejection sampler that simulates a 6-faced die roll (we assume faces are numbered from 0 to 5) with fair coin flips. The naïve implementation would flip three coins, interpret the result as a binary number between 0 and 7, return the result if it is from 0 to 5, and restart the simulation otherwise. A more efficient implementation would use an *early abort* strategy: after observing the first two coin flips, if they are both 1 we can restart without the need for a third coin flip. We implement an early abort rejection sampler *dsim* in Figure 6. We will show that this is contextually equivalent to a uniform die roll, i.e., program *droll* in Figure 6. As an intermediate step, we prove that they are both contextually equivalent to the simple rejection sampler *drej* in the same figure, which samples a uniform number between 0 and 7, returns the result if it is 5 or below, and restarts otherwise.

```
dsim () ≜
  let ι = tape 1 in
  let b2 = rand 1 ι in
  let b1 = rand 1 ι in
  if (b1 == 1 && b2 == 1) then dsim
   else let b0 = rand 1 ι in
        4 ∗ b2 + 2 ∗ b1 + b0
```

```
drej () ≜
  let ι = tape 7 in
  let r = rand 7 ι in
  if (r > 5) then drej
   else r
```

```
droll () ≜
  let ι = tape 5 in
   rand 5 ι
```

Fig. 6. Three algorithms to sample a fair die

The proof thus requires showing the chains of logical refinements $\emptyset \vdash dsim \precsim drej \precsim droll\colon \text{unit} \to \text{nat}$ and $\emptyset \vdash droll \precsim drej \precsim dsim\colon \text{unit} \to \text{nat}$. The proofs of $dsim \precsim drej$ and $drej \precsim dsim$ are mostly symmetric. The proof relies on using wp-rec and the wp-many-to-one coupling rule, to ensure that the three bits we sample to the tape with bound 1 are a binary encoding of the number sampled to the tape with bound 7, and therefore both conditionals resolve to the same branch. In the case they both take the branch with the recursive call, we can apply our inductive hypothesis, otherwise both programs will terminate immediately and return equal values.

The proofs of $drej \precsim droll$ and $droll \precsim drej$ is almost identical to the proofs presented for the rejection samplers (see §4.2), except that we are not only proving that the two programs are equivalent when executed in isolation, but *contextually equivalent* under all contexts using our logical relations. The former of the two uses wp-rec, and our novel rule for fragmented couplings. Note that *droll* only consume a single sample on the tape. The rule for fragmented couplings ensures that we will either sample to the tapes a value above 5 on the left and *nothing* on the right, or we will sample the same value, between 0 and 5, to the tapes on both sides. In the first case, *drej* will consume the value on the tape and call itself recursively, which allows us to use our inductive hypothesis. In the second case, both programs will consume values on their tapes, read the same number, and return equal values.

Finally, we show $droll \precsim drej$. This proof cannot be done by applying wp-rec since the program on the left is not recursive. We will instead use induction by error amplification, through the rule

LOG-IND-ERR. Let us set $k = 4/3$. After applying this rule, we will get ownership of $\lightning(\varepsilon)$ for some arbitrary ε, plus the induction hypothesis

$$\lightning((4/3) \cdot \varepsilon) \mathrel{-\!\!*} (\emptyset \vdash \mathit{droll}\;() \precsim \mathit{drej}\;() : \mathsf{nat}),$$

while our goal becomes

$$\emptyset \vdash \mathit{droll}\;() \precsim \mathit{drej}\;() : \mathsf{nat}\,.$$

We now use our rule for fragmented couplings with errors. This ensures that either (1) we sample to the tapes a value above 5 on the right hand side, nothing on the left, and we amplify our credits by 4/3, or (2) we sample identical values, between 0 and 5, to the tapes on both sides. In the first case, the program on the right hand side will call itself recursively, but now we will own $\varepsilon(4/3) \cdot \varepsilon$, which is precisely what we need to apply our inductive hypothesis and conclude. Otherwise, both values will have the same value on the tapes, and will terminate and return the same result.

While this example is conceptually simple, the reasoning patterns it uses, as well as the different induction principles that we can use depending on the presence or absence of recursion are important subtleties of our approach, and is fundamental to understanding other examples, such as that of the B+ tree in Appendix A.2.

A.2 Sampling from B+ Trees

In this case study, we show the correctness of a rejection sampling scheme developed by Olken and Rotem [1989] for drawing a random sample from a B+ tree. Up to this point, previous examples have made use of only simple forms of state and the contextual equivalences were for simple type signatures. This case study demonstrates how Approxis is able to handle complex mutable state and establish equivalences that rely on type abstraction.

To motivate Olken and Rotem's sampling algorithm, we first summarize some relevant facts about B+ trees. A B+ tree [Bayer and McCreight 1972] is a tree data structure that is widely used for storing data in filesystems and databases. In contrast to a binary search tree, a B+ tree's internal nodes may have more than 2 children. Random sampling from a B+ tree can be used to draw random records from such databases in order to carry out a statistical analysis. Because the tree may store many elements, it is not efficient when drawing a sample to first reprocess the entire database into an alternate representation. Instead, the sample must be drawn working directly over the tree structure.

The sampling algorithm we consider relies on 3 key properties of a B+ tree: (1) data elements are only stored at the leaves of the tree, (2) the height of the tree is perfectly balanced, meaning that the length of the path from the root to a leaf is the same for all leaves, (3) each node has at most M children. Since the algorithm only requires these properties for correctness, our proof will work with trees that are only assumed to satisfy these three properties, instead of assuming all of the invariants of a B+ tree.

Before presenting Olken and Rotem's algorithm, let us first consider a naïve sampling algorithm that will serve as a correctness specification. If we knew that the tree contained N total elements, then one approach to drawing a random sample would be to first generate a random number uniformly from $\{0, \ldots, N-1\}$ and then find and return the i-th element in the tree, numbering the leaves from left to right. This approach correctly produces a uniform sample from the tree, but the challenge lies in efficiently finding the i-th element in the tree. It is easy to find this element if we assume that the tree is a *ranked* B+ tree, where intermediate notes are additionally annotated with the total number of leaves that are descendants of the node. The function naive_sample in Figure 7 implements this algorithm for sampling from a ranked B+ tree. However, maintaining this rank information has a cost: *every* insertion in the tree requires modifying all of the nodes that are

```
naive_sample tree ≜
  let ι = tape (num_leaves tree − 1) in
  let i = rand (num_leaves tree − 1) ι in
  let rec f  t num =
    match t with
    | Lf v ⇒ v
    | Br l ⇒
      let (prev, idx) = search l num in
      let child = l[idx] in
      f (! child) (num − prev)
    end in
  f tree i
```

```
optimized_sample tree ≜
  let rec  draw t ι =
    match t with
    | Lf v ⇒ Some  v
    | Br l ⇒  let idx = rand (M − 1) ι in
              match List.nth l idx with
              | Some child ⇒ draw (! child)
              | None         ⇒ None
              end
    end in
  let rec  f _ =
    let ι = tape (M − 1) in
    match draw tree ι with
    | Some  v ⇒ v
    | None     ⇒ f ()
    end in
  f ()
```

Fig. 7. Naïve algorithm for sampling from a ranked B+ tree and the Olken and Rotem [1989] algorithm for rejection sampling from a non-ranked B+ tree.

ancestors of the inserted node to increase the recorded leaf counts. In contrast, inserting into a (non-ranked) B+ tree, most insertions only require modifying the parent of the inserted element.

Olken and Rotem developed a rejection sampling algorithm for sampling from a non-ranked B+ tree. We call this the optimized algorithm, implemented as optimized_sample in Figure 7. This function makes use of the early abort technique we saw in Appendix A.1. Starting from the root, it samples a random number i uniformly from $\{0, 1, \ldots, M - 1\}$, where M is the maximum number of children a node can have. It selects the i-th child and recurses on it, until it reaches a leaf. If the current node does not have an i-th child, we return to the root and restart the algorithm. The intuition behind the correctness of the optimized algorithm is that it is somewhat similar to sampling random leafs from a full multi-way tree, i.e. a B+ tree where each intermediate node holds M branches. In the case where we walk down a branch that is not present in the original B+ tree, we reject this branch and start all over again.

Our main result for this case study shows that the naïve algorithm and the optimized algorithm are equivalent. Of course, these algorithms are only equivalent when they operate over well-formed trees, so we state this result as a contextual equivalence about two different implementations of an abstract data type with operations for constructing and sampling from the tree. To state this precisely, we first define the following functions (code omitted): init_tree, which takes an integer and returns a B+ tree containing that single integer, insert_tree, which inserts an integer into a tree, and build_ranked, which takes a (non-ranked) B+ tree and returns a ranked B+ tree with the same entries and shape. Next, we define the following two packed tuples that bundle the operations for the B+ tree:

$$\begin{aligned} \text{opt} &\triangleq \textsf{pack}\,(\text{init\_tree}, \text{insert\_tree}, \text{optimized\_sample}) \\ \text{naive} &\triangleq \textsf{pack}\,(\text{init\_tree}, \text{insert\_tree}, \lambda t.\text{naive\_sample}\ (\text{build\_ranked}\ t)) \end{aligned}$$

```
intermediate_sample tree ≜
  let d = get_depth tree in
  let ι = tape (M ˆ d − 1) in
  let rec intermediate_sample′ t =
    let idx = rand (M ˆ d − 1) ι in
    let rec f t num d =
      match t with
      | Lf v ⇒ v
      | Br l ⇒ let idx = num `quot` (M ˆ (d − 1)) in
               match List.nth l idx with
               | Some child ⇒ f (! child) (num − idx · (M ˆ (d − 1))) (d − 1)
               | None       ⇒ intermediate_sample′ tree
               end
      end in
    f t idx d in
  intermediate_sample′ tree
```

Fig. 8. Intermediate algorithm for sampling from a non-ranked B+ tree.

where the sampling routine in naive takes a tree t, builds the ranked version of the tree, and then uses the naïve routine.[4] With these preliminaries in place, our main result can be stated as $\vdash$ opt $\simeq_{\text{ctx}}$ naive : τ, where τ is the following existential type:

$$\tau \triangleq \exists \tau.\, (\mathsf{Int} \to \tau) \times (\tau \times \mathsf{Int} \to 1) \times (\tau \to \mathsf{Int})$$

The proof of this equivalence in our full Coq development is too long to fully explain here. However, at a high level, the proof has two components. First, there is the non-probabilistic reasoning showing that the various routines traverse and modify the trees correctly, *e.g.* that the height-balanced invariant is maintained by insert_tree, or that build_ranked correctly computes ranks. This aspect in fact makes up the bulk of the proof, and consists of using traditional separation-logic style reasoning about trees. For this part, Approxis's support for the rich reasoning principles developed in earlier separation logics is essential.

The second component is the actual probabilistic reasoning using couplings. Here, the coupling reasoning in this proof is quite similar to the arguments used for proving the equivalence of the die sampling routines in Appendix A.1. To make this correspondence clearer, and to simplify the reasoning, we introduce an intermediate sampling routine, intermediate_sample, shown in Figure 8. The intermediate program takes in a tree, computes the depth d of its leaves, and samples a value from rand$(M^d - 1)$. It then interprets this number as a path through the tree. To do so, the program treats it after it were a d digit number written in base M, in which the i-th digit represents a child to select at depth i. If, on reaching depth i it finds that the corresponding child does not exist, then it rejects and repeats with a fresh sample.

We can see then that naive_sample is similar to *droll*: it always succeeds because it samples an index of a valid leaf, just as *droll* always samples a number that is in range. Meanwhile, intermediate_sample is like *drej*, as it samples a large number representing an entire path in a tree, and then rejects if that path is invalid, much as *drej* rejects if its sample is too large.

[4] Of course it would be highly inefficient to construct a ranked tree every time a sample is to be drawn, but naive here serves as a correctness specification for opt, so its efficiency is not relevant.

Finally, optimized_sample is like *dsim*, in that it samples the path layer-by-layer and rejects early if the path is invalid, just as *dsim* samples bit-by-bit and rejects early if the number is already too large. Thus for example, in proving that naive_sample $\precsim$ intermediate_sample, we use fragmented couplings and error amplification, just as we did for *droll* $\precsim$ *drej*, while proving optimized_sample $\precsim$ intermediate_sample, we use the wp-many-to-one and wp-rec rule.

References

Carmine Abate, Philipp G. Haselwarter, Exequiel Rivas, Antoine Van Muylder, Théo Winterhalter, Cătălin Hriţcu, Kenji Maillard, and Bas Spitters. 2021. SSProve: A Foundational Framework for Modular Cryptographic Proofs in Coq. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*. 1–15. https://doi.org/10.1109/CSF51468.2021.00048

Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg, Shin-ya Katsumata, and Tetsuya Sato. 2021. Higher-order probabilistic adversarial computations: categorical semantics and program logics. *Proc. ACM Program. Lang.* 5, ICFP, Article 93 (Aug 2021), 30 pages. https://doi.org/10.1145/3473598

Alejandro Aguirre and Lars Birkedal. 2023. Step-Indexed Logical Relations for Countable Nondeterminism and Probabilistic Choice. *Proc. ACM Program. Lang.* 7, POPL, Article 2 (Jan 2023), 28 pages. https://doi.org/10.1145/3571195

Alejandro Aguirre, Philipp G. Haselwarter, Markus de Medeiros, Kwing Hei Li, Simon Oddershede Gregersen, Joseph Tassarotti, and Lars Birkedal. 2024. Error Credits: Resourceful Reasoning about Error Bounds for Higher-Order Probabilistic Programs. *Proc. ACM Program. Lang.* 8, ICFP, Article 246 (Aug 2024), 33 pages. https://doi.org/10.1145/3674635

José Bacelar Almeida, Denis Firsov, Tiago Oliveira, and Dominique Unruh. 2023. Leakage-Free Probabilistic Jasmin Programs. Cryptology ePrint Archive, Paper 2023/1514. https://eprint.iacr.org/2023/1514 https://eprint.iacr.org/2023/1514.

Robert Atkey. 2011. Amortised Resource Analysis with Separation Logic. *Logical Methods in Computer Science* Volume 7, Issue 2 (June 2011). https://doi.org/10.2168/LMCS-7(2:17)2011

Martin Avanzini, Gilles Barthe, Davide Davoli, and Benjamin Grégoire. 2024. A quantitative probabilistic relational Hoare logic. arXiv:2407.17127 [cs.LO] https://arxiv.org/abs/2407.17127

Jialu Bao, Simon Docherty, Justin Hsu, and Alexandra Silva. 2021. A bunched logic for conditional independence. In *Proceedings of the 36th Annual ACM/IEEE Symposium on Logic in Computer Science* (Rome, Italy) *(LICS '21)*. Association for Computing Machinery, New York, NY, USA, Article 13, 14 pages. https://doi.org/10.1109/LICS52264.2021.9470712

Jialu Bao, Emanuele D'Osualdo, and Azadeh Farzan. 2024. Bluebell: An Alliance of Relational Lifting and Independence For Probabilistic Reasoning. arXiv:2402.18708 [cs.LO] https://arxiv.org/abs/2402.18708

Jialu Bao, Marco Gaboardi, Justin Hsu, and Joseph Tassarotti. 2022. A separation logic for negative dependence. *Proc. ACM Program. Lang.* 6, POPL, Article 57 (Jan 2022), 29 pages. https://doi.org/10.1145/3498719

Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. 2014. *EasyCrypt: A Tutorial.* Springer International Publishing, Cham, 146–166. https://doi.org/10.1007/978-3-319-10082-1_6

Gilles Barthe, Thomas Espitau, Benjamin Grégoire, Justin Hsu, Léo Stefanesco, and Pierre-Yves Strub. 2015. Relational Reasoning via Probabilistic Coupling. In *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*. 387–401. https://doi.org/10.1007/978-3-662-48899-7_27

Gilles Barthe, Thomas Espitau, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2017. Proving expected sensitivity of probabilistic programs. *Proc. ACM Program. Lang.* 2, POPL, Article 57 (Dec 2017), 29 pages. https://doi.org/10.1145/3158145

Gilles Barthe, Noémie Fong, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2016a. Advanced Probabilistic Couplings for Differential Privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. 55–67. https://doi.org/10.1145/2976749.2978391

Gilles Barthe, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2016b. A Program Logic for Union Bounds. In *43rd International Colloquium on Automata, Languages, and Programming (ICALP 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 55)*, Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 107:1–107:15. https://doi.org/10.4230/LIPIcs.ICALP.2016.107

Gilles Barthe, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2016c. Proving Differential Privacy via Probabilistic Couplings. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science* (New York, NY, USA) *(LICS '16)*. Association for Computing Machinery, New York, NY, USA, 749–758. https://doi.org/10.1145/2933575.2934554

Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. 2009. Formal certification of code-based cryptographic proofs. *SIGPLAN Not.* 44, 1 (Jan 2009), 90–101. https://doi.org/10.1145/1594834.1480894

Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. 2010. Programming Language Techniques for Cryptographic Proofs. In *Interactive Theorem Proving*, Matt Kaufmann and Lawrence C. Paulson (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 115–130.

Gilles Barthe, Justin Hsu, and Kevin Liao. 2019. A probabilistic separation logic. *Proc. ACM Program. Lang.* 4, POPL, Article 55 (Dec 2019), 30 pages. https://doi.org/10.1145/3371123

Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. 2012. Probabilistic relational reasoning for differential privacy. *SIGPLAN Not.* 47, 1 (Jan 2012), 97–110. https://doi.org/10.1145/2103621.2103670

Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Thomas Noll. 2019. Quantitative separation logic: a logic for reasoning about probabilistic pointer programs. *Proc. ACM Program. Lang.* 3, POPL, Article 34 (Jan 2019), 29 pages. https://doi.org/10.1145/3290347

Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Lena Verscht. 2023. A Calculus for Amortized Expected Runtimes. *Proc. ACM Program. Lang.* 7, POPL, Article 67 (Jan 2023), 30 pages. https://doi.org/10.1145/3571260

Rudolf Bayer and Edward M. McCreight. 1972. Organization and Maintenance of Large Ordered Indices. *Acta Informatica* 1 (1972), 173–189. https://doi.org/10.1007/BF00288683

Mihir Bellare and Phillip Rogaway. 2004. Code-Based Game-Playing Proofs and the Security of Triple Encryption. *IACR Cryptol. ePrint Arch.* (2004), 331. http://eprint.iacr.org/2004/331

Nick Benton. 2004. Simple relational correctness proofs for static analyses and program transformations. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, Neil D. Jones and Xavier Leroy (Eds.). ACM, 14–25. https://doi.org/10.1145/964001.964003

Aleš Bizjak and Lars Birkedal. 2015. Step-Indexed Logical Relations for Probability. In *Foundations of Software Science and Computation Structures*, Andrew Pitts (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 279–294.

Bruno Blanchet. 2005. A Computationally Sound Automatic Prover for Cryptographic Protocols. In *Workshop on the link between formal and computational models*. Paris, France.

Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. 2015. Coquelicot: A User-Friendly Library of Real Analysis for Coq. *Math. Comput. Sci.* 9, 1 (2015), 41–62. https://doi.org/10.1007/S11786-014-0181-1

Chris Brzuska, Antoine Delignat-Lavaud, Cédric Fournet, Konrad Kohbrok, and Markulf Kohlweiss. 2018. State Separation for Code-Based Game-Playing Proofs. In *Advances in Cryptology – ASIACRYPT 2018* (Cham) *(Lecture Notes in Computer Science)*, Thomas Peyrin and Steven Galbraith (Eds.). Springer International Publishing, 222–249. https://doi.org/10.1007/978-3-030-03332-3_9

Arthur Charguéraud and François Pottier. 2019. Verifying the Correctness and Amortized Complexity of a Union-Find Implementation in Separation Logic with Time Credits. *Journal of Automated Reasoning* 62, 3 (March 2019), 331–365. https://doi.org/10.1007/s10817-017-9431-7

Raphaëlle Crubillé and Ugo Dal Lago. 2017. Metric Reasoning About λ-Terms: The General Case. In *Programming Languages and Systems: 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22–29, 2017, Proceedings* (Uppsala, Sweden). Springer-Verlag, Berlin, Heidelberg, 341–367. https://doi.org/10.1007/978-3-662-54434-1_13

Raphaëlle Crubillé and Ugo Dal Lago. 2015. Metric reasoning about λ-terms: The affine case. In *2015 30th Annual ACM/IEEE Symposium on Logic in Computer Science*. 633–644. https://doi.org/10.1109/LICS.2015.64

Ryan Culpepper and Andrew Cobb. 2017. Contextual Equivalence for Probabilistic Programs with Continuous Random Variables and Scoring. In *Programming Languages and Systems*, Hongseok Yang (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 368–392.

Ankush Das, Di Wang, and Jan Hoffmann. 2023. Probabilistic Resource-Aware Session Types. *Proc. ACM Program. Lang.* 7, POPL, Article 66 (Jan 2023), 32 pages. https://doi.org/10.1145/3571259

Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2021. ReLoC Reloaded: A Mechanized Relational Logic for Fine-Grained Concurrency and Logical Atomicity. *Log. Methods Comput. Sci.* 17, 3 (2021). https://doi.org/10.46298/lmcs-17(3:9)2021

Shafi Goldwasser and Silvio Micali. 1984. Probabilistic Encryption. *J. Comput. Syst. Sci.* 28, 2 (1984), 270–299. https://doi.org/10.1016/0022-0000(84)90070-9

Simon Oddershede Gregersen, Alejandro Aguirre, Philipp G. Haselwarter, Joseph Tassarotti, and Lars Birkedal. 2023. Asynchronous Probabilistic Couplings in Higher-Order Separation Logic. *CoRR* abs/2301.10061 (2023). https://doi.org/10.48550/ARXIV.2301.10061 arXiv:2301.10061

Simon Oddershede Gregersen, Alejandro Aguirre, Philipp G. Haselwarter, Joseph Tassarotti, and Lars Birkedal. 2024. Asynchronous Probabilistic Couplings in Higher-Order Separation Logic. *Proc. ACM Program. Lang.* 8, POPL (2024), 753–784. https://doi.org/10.1145/3632868

Chris Hall, David A. Wagner, John Kelsey, and Bruce Schneier. 1998. Building PRFs from PRPs. In *Advances in Cryptology - CRYPTO '98, 18th Annual International Cryptology Conference, Santa Barbara, California, USA, August 23-27, 1998, Proceedings*. 370–389. https://doi.org/10.1007/BFB0055742

Philipp G. Haselwarter, Kwing Hei Li, Markus de Medeiros, Simon Oddershede Gregersen, Alejandro Aguirre, Joseph Tassarotti, and Lars Birkedal. 2024. Tachis: Higher-Order Separation Logic with Credits for Expected Costs. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 313 (Oct. 2024), 30 pages. https://doi.org/10.1145/3689753

Philipp G. Haselwarter, Exequiel Rivas, Antoine Van Muylder, Théo Winterhalter, Carmine Abate, Nikolaj Sidorenco, Cătălin Hriţcu, Kenji Maillard, and Bas Spitters. 2023. SSProve: A Foundational Framework for Modular Cryptographic Proofs in Coq. *ACM Trans. Program. Lang. Syst.* 45, 3, Article 15 (July 2023), 61 pages. https://doi.org/10.1145/3594735

Martin Hofmann and Steffen Jost. 2003. Static prediction of heap space usage for first-order functional programs. *SIGPLAN Not.* 38, 1 (Jan 2003), 185–197. https://doi.org/10.1145/640128.604148

Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*. 256–269. https://doi.org/10.1145/2951913.2951943

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. 2016. Weakest Precondition Reasoning for Expected Run–Times of Probabilistic Programs. In *Programming Languages and Systems*, Peter Thiemann (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 364–389.

Jonathan Katz and Yehuda Lindell. 2021. *Introduction to Modern Cryptography* (third ed.). CRC Press.

John M. Li, Amal Ahmed, and Steven Holtzen. 2023. Lilac: A Modal Separation Logic for Conditional Probability. *Proc. ACM Program. Lang.* 7, PLDI, Article 112 (Jun 2023), 24 pages. https://doi.org/10.1145/3591226

T. Lindvall. 2002. *Lectures on the Coupling Method*. Dover Publications, Incorporated.

Kenji Maillard, Catalin Hritcu, Exequiel Rivas, and Antoine Van Muylder. 2020. The next 700 relational program logics. *Proc. ACM Program. Lang.* 4, POPL (2020), 4:1–4:33. https://doi.org/10.1145/3371072

Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2019. Time Credits and Time Receipts in Iris. In *European Symposium on Programming (Lecture Notes in Computer Science, Vol. 11423)*. Springer, Prague, Czech Republic, 3–29. https://doi.org/10.1007/978-3-030-17184-1_1

Carroll Morgan, Annabelle McIver, and Karen Seidel. 1996. Probabilistic predicate transformers. *ACM Trans. Program. Lang. Syst.* 18, 3 (May 1996), 325–353. https://doi.org/10.1145/229542.229547

Van Chan Ngo, Quentin Carbonneaux, and Jan Hoffmann. 2018. Bounded expectations: resource analysis for probabilistic programs. *SIGPLAN Not.* 53, 4 (Jun 2018), 496–512. https://doi.org/10.1145/3296979.3192394

Frank Olken and Doron Rotem. 1989. Random Sampling from B+ Trees. In *Proceedings of the Fifteenth International Conference on Very Large Data Bases, August 22-25, 1989, Amsterdam, The Netherlands*. 269–277. http://www.vldb.org/conf/1989/P269.PDF

François Pottier, Armaël Guéneau, Jacques-Henri Jourdan, and Glen Mével. 2024. Thunks and Debits in Separation Logic with Time Credits. *Proc. ACM Program. Lang.* 8, POPL, Article 50 (Jan. 2024), 27 pages. https://doi.org/10.1145/3632892

Mike Rosulek. 2021. *The Joy of Cryptography*.

Tetsuya Sato. 2016. Approximate Relational Hoare Logic for Continuous Random Samplings. In *The Thirty-second Conference on the Mathematical Foundations of Programming Semantics, MFPS 2016, Carnegie Mellon University, Pittsburgh, PA, USA, May 23-26, 2016*. 277–298. https://doi.org/10.1016/J.ENTCS.2016.09.043

Victor Shoup. 2004. Sequences of Games: A Tool for Taming Complexity in Security Proofs. eprint:2004.332 https://eprint.iacr.org/2004/332

The Coq Development Team. 2024. *The Coq Proof Assistant*. https://doi.org/10.5281/zenodo.11551307

Hermann Thorisson. 2000. *Coupling, stationarity, and regeneration*. Springer-Verlag, New York. xiv+517 pages.

Amin Timany, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2024. A Logical Approach to Type Soundness. *J. ACM* (Jul 2024). https://doi.org/10.1145/3676954 Just Accepted.

Aaron Turon, Derek Dreyer, and Lars Birkedal. 2013. Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*. 377–390. https://doi.org/10.1145/2500365.2500600

C. Villani. 2008. *Optimal Transport: Old and New*. Springer Berlin Heidelberg.

Mitchell Wand, Ryan Culpepper, Theophilos Giannakopoulos, and Andrew Cobb. 2018. Contextual equivalence for a probabilistic language with continuous random variables and recursion. *Proc. ACM Program. Lang.* 2, ICFP, Article 87 (Jul 2018), 30 pages. https://doi.org/10.1145/3236782

Di Wang, David M. Kahn, and Jan Hoffmann. 2020. Raising expectations: automating expected cost analysis with types. *Proc. ACM Program. Lang.* 4, ICFP, Article 110 (Aug 2020), 31 pages. https://doi.org/10.1145/3408992

Yizhou Zhang and Nada Amin. 2022. Reasoning about "reasoning about reasoning": semantics and contextual equivalence for probabilistic programs with nested queries and recursion. *Proc. ACM Program. Lang.* 6, POPL, Article 16 (Jan 2022), 28 pages. https://doi.org/10.1145/3498677

Received 2024-07-11; accepted 2024-11-07