Bluebell: An Alliance of Relational Lifting and Independence for Probabilistic Reasoning

JIALU BAO, Cornell University, NY, US
EMANUELE D'OSUALDO, MPI-SWS, Germany and University of Konstanz, Germany
AZADEH FARZAN, University of Toronto, Canada

We present Bluebell, a program logic for reasoning about probabilistic programs where unary and relational styles of reasoning come together to create new reasoning tools. Unary-style reasoning is very expressive and is powered by foundational mechanisms to reason about probabilistic behavior like *independence* and *conditioning*. The relational style of reasoning, on the other hand, naturally shines when the properties of interest *compare* the behavior of similar programs (e.g. when proving differential privacy) managing to avoid having to characterize the output distributions of the individual programs. So far, the two styles of reasoning have largely remained separate in the many program logics designed for the deductive verification of probabilistic programs. In Bluebell, we unify these styles of reasoning through the introduction of a new modality called "joint conditioning" that can encode and illuminate the rich interaction between *conditional independence* and *relational liftings*; the two powerhouses from the two styles of reasoning.

CCS Concepts: • **Theory of computation → Separation logic**; **Logic and verification**; **Probabilistic computation**; **Program verification**; • **Software and its engineering** → *Software verification.*

Additional Key Words and Phrases: Deductive Verification, Relational Logic, Conditional Independence, Relational Lifting, Weakest Precondition

ACM Reference Format:
Jialu Bao, Emanuele D'Osualdo, and Azadeh Farzan. 2025. Bluebell: An Alliance of Relational Lifting and Independence for Probabilistic Reasoning. *Proc. ACM Program. Lang.* 9, POPL, Article 58 (January 2025), 31 pages. https://doi.org/10.1145/3704894

1 INTRODUCTION

Probabilistic programs are pervasive, appearing as machine learned subsystems, implementations of randomized algorithms, cryptographic protocols, and differentially private components, among many more. Ensuring reliability of such programs requires formal frameworks in which correctness requirements can be formalized and verified for such programs. Similar to the history of classical program verification, a lot of progress in this has come in the form of program logics for probabilistic programs. In the program logic literature, there are two main styles of reasoning for probabilistic programs: *unary* and *relational*, depending on the nature of the property of interest. For instance, for differential privacy or cryptographic protocols correctness, the property of interest is naturally expressible relationally. In contrast, for example, specifying the expected cost of a randomized algorithm is naturally done in the unary style.

Authors' addresses: Jialu Bao, jb965@cornell.edu, Cornell University, Ithaca, NY, US; Emanuele D'Osualdo, emanuele.dosualdo@uni-konstanz.de, MPI-SWS, Saarland Informatics Campus, Germany and University of Konstanz, Germany; Azadeh Farzan, azadeh@cs.toronto.edu, University of Toronto, Toronto, Canada.

2475-1421/2025/1-ART58
https://doi.org/10.1145/3704894

Unary goals are triples $\{P\}\ t\ \{Q\}$ where t is a probabilistic program, P and Q are the pre- and post-conditions, i.e. *predicates over distributions of stores*. Such triples assert that running t on an input store drawn from a distribution satisfying P results in a distribution over output stores which satisfies Q. Unary reasoning for probabilistic programs has made great strides, producing logics for reasoning about expectations [Aguirre et al. 2021; Kaminski 2019; Kaminski et al. 2016; Kozen 1983; Moosbrugger et al. 2022; Morgan et al. 1996] and probabilistic independence [Barthe et al. 2019]. DIBI [Bao et al. 2021] and Lilac [Li et al. 2023a], which are the most recent, made a strong case for adding power to reason about conditioning and independence in one logic. Intuitively, conditioning on some random variable x allows to focus on the distribution of other variables assuming x is some deterministic outcome v; two variables are (conditionally) independent if knowledge of one does not give any knowledge of the other (under conditioning). Lilac argued for (conditional) independence as the fundamental source of modularity in the probabilistic setting.

Relational goals, in contrast, specify a desired relation between the output distributions of *two* programs t_1 and t_2, for example, that t_1 and t_2 produce the same output distribution. In principle, proving such goals can be approached in a unary style: if the output distributions can be characterized individually for each program, then they can be compared after the fact. More often than not, however, precisely characterizing the output distribution of a program can be extremely challenging. Relational program logics like pRHL [Barthe et al. 2009] and its successors [Aguirre et al. 2017; Barthe et al. 2015, 2009; Gregersen et al. 2024; Hsu 2017], allow for a different and often more advantageous strategy. The idea is to consider the two programs side by side, and analyse their code as if executed in lockstep. If the effect of a step on one side is "matched" by the corresponding step on the other side, then the overall outputs would also "match". This way, the proof only shows that whatever is computed on one side, will be matched by a computation on the other, without having to characterise what the actual output is.

This idea of "matching" probabilistic steps is formalised in these logics via the notion of *couplings* [Barthe et al. 2015, 2009]. The two programs can be conceptually considered to execute in two "parallel universes", where they are oblivious to each other's randomness. It is therefore sound to pretend their executions draw samples from a common source of randomness (called a *coupling*) in any way that eases the argument, as long as the marginal distribution of the correlated runs in each universe coincides with the original one. For example, if both programs flip a fair coin, one can force the outcomes of the coin flips to be the same (or the opposite of each other, depending on which serves the particular line of argument better). Relating the samples in a specific way helps with relating the distributions step by step, to support a relational goal. Couplings, when applicable, permit relational logics to elegantly sidestep the need to characterize the output distributions precisely. As such, relational logics hit an ergonomic sweet spot in reasoning style by restricting the form of the proofs that can be carried out.

Consider, for example, the code in Fig. 1. The `BelowMax`(x, S) procedure takes N samples from a non-empty set $S \subseteq \mathbb{Z}$, according to an (arbitrary) distribution $\mu_S : \mathbb{D}(S)$; if any of the samples is larger than the given input x it declares x to be below the maximum of S. The `AboveMin`(x, S) approximates in the same way whether x is above the minimum of S. These are Monte Carlo style algorithms with a *false bias*: if the answer is false, they always correctly produce it, and if the answer is true, then they correctly classify it with a probability that depends on N (i.e., the number of samples). It is a well-known fact that Monte Carlo style algorithms can be composed. For example, `BETW_SEQ` runs `BelowMax`(x, S) and `AboveMin`(x, S) to produce a *false-biased* Monte Carlo algorithm for approximately deciding whether x lies *within* the extrema of S. Now, imagine a programmer proposed `BETW`, as a way of getting more mileage out of the number of samples drawn; both procedures take $2N$ samples, but `BETW` performs more computation for each sample. Such optimisations are not really concerned about what the precise output distribution of each

```
def BelowMax($x$,$S$):
  repeat $N$:
    q :≈ $\mu_S$
    r := r || q ≥ $x$
```

```
def AboveMin($x$,$S$):
  repeat $N$:
    p :≈ $\mu_S$
    l := l || p ≤ $x$
```

```
def BETW_SEQ($x$, $S$):
  BelowMax($x$,$S$);
  AboveMin($x$,$S$);
  d := r && l
```

```
def BETW($x$,$S$):
  repeat $2N$:
    s :≈ $\mu_S$
    l := l || s ≤ $x$
    r := r || s ≥ $x$
  d := r && l
```

Fig. 1. A stochastic dominance example: composing Monte Carlo algorithms two different ways. $N \in \mathbb{N}$ is some fixed constant, and all variables are initially 0.

code is, but rather that a *true* answer is produced with higher probability by BETW; in other words, its *stochastic dominance* over BETW_SEQ.

A unary program logic has only one way of reasoning about this type of stochastic-dominance: it has to analyze each code in isolation, characterize its output distribution, and finally assert/prove that one dominates the other. In contrast, there is a natural *relational strategy* for proving this goal: we can match the N samples of BelowMax with N of the samples of BETW, and the N samples of AboveMin with the remaining samples of BETW in lockstep, and for each of these aligned steps, BETW has more chances of turning l and r to 1 (and they can only increase).

Unary logics can express information about distributions with arbitrary levels of precision; yet none can encode the simple natural proof idea outlined above. This suggests an opportunity: Bring native relational reasoning support to an expressive unary logic, like Lilac. Such a logic can be based on assertions over distributions, thus able to be as precise and expressive as unary logics, yet it can support relational reasoning natively and as such can encode the argument outlined above at the appropriate level of abstraction. To explore this idea, let us outline the basic principle that we would need to import from relational reasoning: *relational lifting*.

Relational logics use variants of judgments of the form $\{R_1\}[1{:}\, t_1, 2{:}\, t_2]\{R_2\}$, where t_1 and t_2 are the two programs we are comparing and R_1 and R_2 are the relational pre- and post-conditions. R_1 and R_2 differ from unary assertions in two ways: first they are used to relate two distributions instead of constraining a single one. Second, they are predicates over *pairs of stores*, and not of distributions directly. Let us call predicates of this type “deterministic relations”. If R was a deterministic predicate over a single store, requiring it to hold with probability 1 would naturally lift it to a predicate $\lceil R \rceil$ over distributions of stores. When R is a deterministic relation between pairs of stores, its *relational lifting* $\lfloor R \rfloor$ relates two distributions over stores $\mu_1, \mu_2 : \mathbb{D}(\mathbb{S})$, if (1) there is a distribution over *pairs* of stores $\mu : \mathbb{D}(\mathbb{S} \times \mathbb{S})$ such that its marginal distributions on the first and second store coincide with μ_1 and μ_2 respectively, (i.e. μ is a *coupling* of μ_1 and μ_2) and (2) μ samples pairs of stores satisfying the relation R with probability 1. Such relational liftings can encode a variety of useful relations between distributions. For instance, let $R = (\mathsf{x}\langle 1 \rangle = \mathsf{x}\langle 2 \rangle)$ relate stores s_1 and s_2 if they both assign the same value to x; then the lifting $\lfloor R \rfloor$ holds for two distributions $\mu_1, \mu_2 : \mathbb{D}(\mathbb{S})$ if and only if they induce the same distributions in x. Similarly, the lifting $\lfloor \mathsf{x}\langle 1 \rangle \leq \mathsf{x}\langle 2 \rangle \rfloor$ encodes stochastic dominance of the distribution of x in μ_2 over the one in μ_1.

Relational proofs built out of relational lifting then work by using deterministic relations as assertions, and showing that a suitably coupled lockstep execution of the two programs satisfies each assertion with probability 1. To bring relational reasoning to unary logics, we want to preserve the fact that assertions are over distributions, and yet support relational lifting as the key abstraction to do relational reasoning. This new logic can equally be viewed as a relational logic with assertions over pairs of distributions (rather than over pairs of stores). With such a view, seeing relational lifting as one among many constructs to build assertions seems like a very natural, yet completely unexplored, idea.

What is entirely non-obvious is whether relational lifting works well as an abstraction together with the other key "unary" constructs, such as independence and conditioning, that are the source of expressive power of unary logics. For example, from the properties of couplings, we know that establishing $\lfloor \mathsf{x}\langle 1\rangle = \mathsf{x}\langle 2\rangle \rfloor$ implies that $\mathsf{x}\langle 1\rangle$ and $\mathsf{x}\langle 2\rangle$ are identically distributed; this can be expressed as an entailment:

$$\lfloor \mathsf{x}\langle 1\rangle = \mathsf{x}\langle 2\rangle \rfloor \dashv\vdash \exists \mu.\, \mathsf{x}\langle 1\rangle \sim \mu \land \mathsf{x}\langle 2\rangle \sim \mu \tag{1}$$

The equivalence says that establishing a coupling that can (almost surely) equate the values of $\mathsf{x}\langle 1\rangle$ and $\mathsf{x}\langle 2\rangle$, amounts to establishing that the two variables are identically distributed. The equivalence can be seen as a way to interface "unary" facts and relational liftings.

Probability theory is full of lemmas of this sort and it is clearly undesirable to admit any lemma that is needed for one proof or another as an axiom in the program logic. Can we have a logic in which they are derivable without having to abandon its nice abstractions? Can the two styles be interoperable at the level of the logic? In this paper, we provide an affirmative answer to this question by proposing a new program logic called Bluebell.

We propose that relational lifting does in fact have non-trivial and useful interactions with independence and conditioning. Remarkably, Bluebell's development is unlocked by a more fundamental observation: once an appropriate notion of conditioning is defined in Bluebell, relational lifting and its laws can be derived from this foundational conditioning construct.

The key idea is a new characterization of relational lifting as a form of conditioning: whilst relational lifting is usually seen as a way to induce a relation over distributions from a deterministic relation, Bluebell sees it as a way to go from a tuple of distributions to a relation between the values of some conditioned variables. More precisely:

- We introduce a new *joint conditioning* modality in Bluebell which can be seen, in hindsight, as a natural way to condition when dealing with tuples of distributions.
- We show that joint conditioning can represent uniformly both, conditioning *à la* Lilac, and relational lifting as derived notions in Bluebell.
- We prove a rich set of general rules for joint conditioning, from which we can derive both known and novel proof principles for conditioning and for relational liftings in Bluebell.

Interestingly, our joint conditioning modality can replicate the same reasoning style of Lilac's modality, while having a different semantics (and validating an overlapping but different set of rules as a result). This deviation in the semantics is a stepping stone for obtaining an adequate generalization to the n-ary case (unifying unary and binary as special cases). We expand on these ideas in Section 2, using a running example. More importantly, our joint conditioning enables Bluebell to

- Accommodate unary and relational reasoning in a fundamentally interoperable way: For instance, we showcase the interaction between lifting and conditioning in the derivation of our running example in Section 2.
- Illuminate known reasoning principles: For instance, we discuss how Bluebell emulates pRHL-style reasoning in Section 5.1.
- Propose new tools to build program proofs: For instance, we discuss out-of-order coupling of samples through seq-swap in Section 2.4.
- Enable the exploration of the theory of high-level constructs like relational lifting (via the laws of independence and joint conditioning): For instance, novel broadly useful rules rl-merge and rl-convex, discussed in Section 2 can be derived within Bluebell.

All proofs, omitted details, and additional examples can be found in [Bao et al. 2024].

2 A TOUR OF BLUEBELL

In this section we will highlight the main key ideas behind Bluebell, using a running example.

2.1 The Alliance

We work with a first-order imperative probabilistic programming language consisting of programs $t \in \mathbb{T}$ that mutate a variable store $s \in \mathbb{S}$ (i.e. a finite map from variable names $\mathbb{X}$ to values $\mathbb{V}$). We only consider discrete distributions (but with possibly infinite support). In Fig. 2 we show a simple example adapted from [Barthe et al. 2019]: the `encrypt` procedure uses a fair coin flip to generate an encryption key `k`, generates a plaintext message in boolean variable `m` (using a coin flip with some bias p) and produces the ciphertext `c` by XORing the key and the message. A desired property of the program is that the ciphertext should be indistinguishable from an unbiased coin flip; as a binary triple:

```
def encrypt():
  k :≈ Ber(½)
  m :≈ Ber(p)
  c := k xor m
```

Fig. 2. One time pad.

$$\{\mathtt{True}\}[1{:}\,\mathtt{encrypt()}, 2{:}\,\mathtt{c} :\approx \mathtt{Ber}(\tfrac{1}{2})]\{\lfloor \mathtt{c}\langle 1\rangle = \mathtt{c}\langle 2\rangle \rfloor\} \tag{2}$$

where we use the $\langle i\rangle$ notation to indicate the index of the program store that an expression references. In Section 5.2, we discuss a unary-style proof of this goal in Bluebell. Here, we focus on a relational argument, as a running example. The natural (relational) argument goes as follows. When computing the final XOR, if $\mathtt{m} = 0$ then $\mathtt{c} = \mathtt{k}$, and if $\mathtt{m} = 1$ then $\mathtt{c} = \neg\mathtt{k}$. Since both $\mathtt{k}\langle 1\rangle$ and $\mathtt{c}\langle 2\rangle$ are distributed as *unbiased* coins, they can be coupled either so that they get the same value, or so that they get opposite values (the marginals are the same). One or the other coupling must be established *conditionally* on $\mathtt{m}\langle 1\rangle$, to formalize this argument. Doing so in pRHL faces the problem that the logic is too rigid to permit one to condition on $\mathtt{m}\langle 1\rangle$ before $\mathtt{k}\langle 1\rangle$ is sampled; rather it forces one to establish a coupling of $\mathtt{k}\langle 1\rangle$ and $\mathtt{c}\langle 2\rangle$ right when the two samplings happen. This rigidity is a well-known limitation of relational logics, which we overcome by "immersing" relational lifting in a logic with assertions on distributions. Recent work [Gregersen et al. 2024] proposed workarounds based on ghost code for pre-sampling (see Section 6). We present a different solution based on framing, to the generic problem of out-of-order coupling, in Section 2.4.

Unconstrained by the pRHL assumption that every assertion has to be represented as a relational lifting, we observe three crucial components in the proof idea:

(1) *Probabilistic independence* between the sampling of $\mathtt{k}\langle 1\rangle$ and $\mathtt{m}\langle 1\rangle$, which makes conditioning on $\mathtt{m}\langle 1\rangle$ preserve the distribution of $\mathtt{k}\langle 1\rangle$;
(2) *Conditioning* to perform case analysis on the possible values of $\mathtt{m}\langle 1\rangle$;
(3) *Relational lifting* to represent the existence of couplings imposing the desired correlation between $\mathtt{k}\langle 1\rangle$ and $\mathtt{c}\langle 2\rangle$.

Unary logics like Probabilistic Separation Logics (PSL) [Barthe et al. 2019] and Lilac explored how probabilistic independence can be represented as *separating conjunction*, obtaining remarkably expressive and elegant reasoning principles. In Bluebell, we import the notion of independence from Lilac: Bluebell's assertions are interpreted over tuples of probability spaces $\mathcal{P}$, and $Q_1 * Q_2$ holds on $\mathcal{P}$ if $\mathcal{P}(i)$ can be seen as the *independent product* of $\mathcal{P}_1(i)$ and $\mathcal{P}_2(i)$, for each i, such that the tuples $\mathcal{P}_1$ and $\mathcal{P}_2$ satisfy Q_1 and Q_2 respectively. This means that $\mathtt{x}\langle 1\rangle \sim \mu * \mathtt{y}\langle 1\rangle \sim \mu$ states that $\mathtt{x}\langle 1\rangle$ and $\mathtt{y}\langle 1\rangle$ are independent and identically distributed, as opposed to $\mathtt{x}\langle 1\rangle \sim \mu \land \mathtt{y}\langle 1\rangle \sim \mu$ which merely declares the two variables as identically distributed (but possibly correlated). For a unary predicate over stores R we write $\lceil R\langle i\rangle \rceil$ to mean that the predicate R holds with probability 1 in the distribution at index i.

With these tools it is easy to get through the first two assignments of `encrypt` and the one on component 2 and get to a state satisfying the assertion

$$P = \mathtt{k}\langle 1\rangle \sim \mathsf{Ber}_{\frac{1}{2}} * \mathtt{m}\langle 1\rangle \sim \mathsf{Ber}_{p} * \mathtt{c}\langle 2\rangle \sim \mathsf{Ber}_{\frac{1}{2}} \tag{3}$$

The next ingredient we need is conditioning. We introduce a new modality $\boldsymbol{C}_{\mu}$ for conditioning, in the spirit of Lilac. The modality takes the form $\boldsymbol{C}_{\mu}\, v.\, K(v)$ where μ is a distribution, v is a logical variable bound by the modality (ranging over the support of μ), and $K(v)$ is a family of assertions indexed by v. Before exploring the general meaning of the modality (which we do in Section 2.2), let us illustrate how we would represent conditioning on $\mathtt{m}\langle 1\rangle$ in our running example. We know $\mathtt{m}\langle 1\rangle$ is distributed as Ber_p; conditioning on $\mathtt{m}\langle 1\rangle$ in Bluebell would give us an assertion of the form $\boldsymbol{C}_{\mathsf{Ber}_p}\, v.\, K(v)$ where v ranges over $\{0,1\}$, and the assertions $K(0) = \lceil \mathtt{m}\langle 1\rangle = 0\rceil * P_0$ and $K(1) = \lceil \mathtt{m}\langle 1\rangle = 1\rceil * P_1$ (for some P_0, P_1) represent the properties of the distribution conditioned on $\mathtt{m}\langle 1\rangle = 0$ and $\mathtt{m}\langle 1\rangle = 1$, respectively. Intuitively, the assertion states that the current distribution satisfies $K(v)$ when conditioned on $\mathtt{m}\langle 1\rangle = v$. Semantically, a distribution μ_0 satisfies the assertion $\boldsymbol{C}_{\mathsf{Ber}_p}\, v.\, K(v)$ if there exists distributions κ_0, κ_1 such that κ_0 satisfies $K(0)$, κ_1 satisfies $K(1)$, and μ_0 is the convex combination $\mu_0 = p \cdot \kappa_1 + (1-p)\cdot \kappa_0$. When $K(v)$ constrains, as in our case, the value of a variable (here $\mathtt{m}\langle 1\rangle$) to be v, the only κ_0 and κ_1 satisfying the above constraints are the distribution μ_0 conditioned on the variable being 0 and 1 respectively.

Combining independence and conditioning with the third ingredient, relational lifting $\lfloor R\rfloor$ (we discuss more about how to define it in Section 2.2), we can now express with an assertion the desired conditional coupling we foreshadowed in the beginning:

$$Q = \boldsymbol{C}_{\mathsf{Ber}_p}\, v. \left(\lceil \mathtt{m}\langle 1\rangle = v\rceil * \begin{cases} \lfloor \mathtt{k}\langle 1\rangle = \mathtt{c}\langle 2\rangle \rfloor & \text{if } v = 0 \\ \lfloor \mathtt{k}\langle 1\rangle = \neg\mathtt{c}\langle 2\rangle \rfloor & \text{if } v = 1 \end{cases} \right) \tag{4}$$

The idea is that we first condition on $\mathtt{m}\langle 1\rangle$ so that we can see it as the deterministic value v, and then we couple $\mathtt{k}\langle 1\rangle$ and $\mathtt{c}\langle 2\rangle$ differently depending on v.

To carry out the proof idea formally, we are left with two subgoals. The first is to formally prove the entailment $P \vdash Q$. Then, it is possible to prove that after the final assignment to `c` at index 1, the program is in a state that satisfies $Q * \lceil \mathtt{c}\langle 1\rangle = \mathtt{k}\langle 1\rangle \mathbin{\mathtt{xor}} \mathtt{m}\langle 1\rangle\rceil$. To finish the proof we would need to prove that $Q * \lceil \mathtt{c}\langle 1\rangle = \mathtt{k}\langle 1\rangle \mathbin{\mathtt{xor}} \mathtt{m}\langle 1\rangle\rceil \vdash \lfloor \mathtt{c}\langle 1\rangle = \mathtt{c}\langle 2\rangle\rfloor$. These missing steps need laws governing the interaction among independence, conditioning and relational lifting in this n-ary setting.

> A crucial observation of Bluebell is that, by choosing an appropriate definition for the joint conditioning modality $\boldsymbol{C}_{\mu}$, relational lifting can be encoded as a form of conditioning. Consequently, the laws governing relational lifting can be derived from the more primitive laws for joint conditioning. Moreover, the interactions between relational lifting and independence can be derived through the primitive laws for the interactions between joint conditioning and independence.

2.2 Joint Conditioning and Relational Lifting

Now we introduce the *joint* conditioning modality and its general n-ary version. Given $\mu : \mathbb{D}(A)$ and a function $\kappa : A \to \mathbb{D}(B)$ (called a *Markov kernel*), define the distribution $\mathbf{bind}(\mu, \kappa) : \mathbb{D}(B)$ as $\mathbf{bind}(\mu,\kappa) = \boldsymbol{\lambda} b.\, \sum_{a\in A} \mu(a)\cdot \kappa(a)(b)$ and $\mathbf{return}(v) = \delta_v$. The **bind** operation represents a convex combination with coefficients in μ, while δ_v is the Dirac distribution, which assigns probability 1 to the outcome v. These operations form a monad with the distribution functor $\mathbb{D}(\,\cdot\,)$, a special case of the Giry monad [Giry 1982]. Given a distribution $\mu : \mathbb{D}(A)$, and a predicate $K(a)$ over pairs of distributions parametrized by values $a \in A$, we define $\boldsymbol{C}_{\mu}\, a.\, K(a)$ to hold on some (μ_1, μ_2) if

$$\exists \kappa_1, \kappa_2.\, \forall i \in \{1,2\}.\, \mu_i = \mathbf{bind}(\mu, \kappa_i) \land \forall a \in \operatorname{supp}(\mu).\, K(a) \text{ holds on } (\kappa_1(a), \kappa_2(a))$$

Namely, we decompose the pair (μ_1, μ_2) component wise into convex compositions of μ and some kernels κ_1, κ_2, one per component. Then for every a with non-zero probability in μ, we require the predicate $K(a)$ to hold for the pair of distributions $(\kappa_1(a), \kappa_2(a))$. The definition naturally extends to any number of indices.

One powerful application of the joint conditioning modality is to encode relational liftings $\lfloor R \rfloor$. Imagine we want to express $\lfloor \mathsf{k}\langle 1\rangle = \mathsf{c}\langle 2\rangle \rfloor$. It suffices to assert that there exists some distribution $\mu : \mathbb{D}(\mathbb{V} \times \mathbb{V})$ over pairs of values such that $\boldsymbol{C}_\mu(v_1, v_2).\big(\lceil \mathsf{k}\langle 1\rangle = v_1 \rceil \land \lceil \mathsf{c}\langle 2\rangle = v_2 \rceil \land \ulcorner v_1 = v_2 \urcorner\big)$, where $\ulcorner \varphi \urcorner$ denote the embedding of a pure fact φ (i.e. a meta-level statement) into the logic. Let us digest the formula step-by-step. $\boldsymbol{C}_\mu(v_1, v_2).\big(\lceil \mathsf{k}\langle 1\rangle = v_1 \rceil \land \lceil \mathsf{c}\langle 2\rangle = v_2 \rceil\big)$ conditions the distribution at index 1 on $\mathsf{k}\langle 1\rangle = v_1$ and conditions the distribution at index 2 on $\mathsf{c}\langle 2\rangle = v_2$; such simultaneous conditioning is possible only if μ projected to its first index, $\mu \circ \pi_1^{-1}$, is the marginal distribution of $\mathsf{k}\langle 1\rangle$ and μ projected to its second index, $\mu \circ \pi_2^{-1}$, is the marginal distribution of $\mathsf{c}\langle 2\rangle$. Thus, μ is a joint distribution – a.k.a. coupling – of the marginal distributions of $\mathsf{k}\langle 1\rangle$ and $\mathsf{c}\langle 2\rangle$. The full assertion $\boldsymbol{C}_\mu(v_1, v_2).\,(\lceil \mathsf{k}\langle 1\rangle = v_1 \rceil \land \lceil \mathsf{c}\langle 2\rangle = v_2 \rceil \land \ulcorner v_1 = v_2 \urcorner)$ ensures that the coupling μ has non-zero probabilities only on pairs (v_1, v_2) where $v_1 = v_2$, and this is exactly the requirement of the relational lifting $\lfloor \mathsf{k}\langle 1\rangle = \mathsf{c}\langle 2\rangle \rfloor$.

The idea generalizes to arbitrary relation lifting $\lfloor R \rfloor$, which are encoded using assertions of the form $\exists \mu.\, \boldsymbol{C}_\mu(\vec{v}_1, \vec{v}_2).\big(\lceil \vec{\mathsf{x}}\langle 1\rangle = \vec{v}_1 \rceil \land \lceil \vec{\mathsf{x}}\langle 2\rangle = \vec{v}_2 \rceil \land \ulcorner R(\vec{v}_1, \vec{v}_2) \urcorner\big)$. The encoding hinges on the crucial decision in the design of the joint conditioning modality of using the same distribution μ to decompose the distributions at all indices. Then, the assertion inside the conditioning can force μ to be a joint distribution of (marginal) distributions of program states at different indices; and it can further force μ to have non-zero probability only on pairs of program states that satisfy the relation R.

> The remarkable fact is that our formulation of relational lifting directly explains:
> (1) How the relational lifting can be *established*: that is, by providing some joint distribution μ for $\mathsf{k}\langle 1\rangle$ and $\mathsf{c}\langle 2\rangle$ ensuring R (the relation being lifted) holds for their joint outcomes; and
> (2) How the relational lifting can be *used* in entailments: that is, it guarantees that if one conditions on the store, R holds between the (now deterministic) variables.

To make these definitions and connections come to fruition we need to study which laws are supported by the joint conditioning modality and whether they are expressive enough to reason about distributions pairs without having to drop down to the level of semantics.

2.3 The Laws of Joint Conditioning

We survey the key laws for joint conditioning in this section, and explore a vital consequence of defining both conditional independence and relational lifting based on the joint conditioning modality: the laws of both can be derived from a set of expressive laws about joint conditioning alone. To keep the exposition concrete, we focus on a small subset of laws that are enough to prove the example of Section 2.1. Let us focus first on proving:

$$\mathsf{k}\langle 1\rangle \sim \mathsf{Ber}_{\frac{1}{2}} * \mathsf{m}\langle 1\rangle \sim \mathsf{Ber}_p * \mathsf{c}\langle 2\rangle \sim \mathsf{Ber}_{\frac{1}{2}} \vdash \boldsymbol{C}_{\mathsf{Ber}_p}\, v. \left(\lceil \mathsf{m}\langle 1\rangle = v \rceil * \begin{cases} \lfloor \mathsf{k}\langle 1\rangle = \mathsf{c}\langle 2\rangle \rfloor & \text{if } v = 0 \\ \lfloor \mathsf{k}\langle 1\rangle = \neg\mathsf{c}\langle 2\rangle \rfloor & \text{if } v = 1 \end{cases} \right) \tag{5}$$

We need the following primitive laws of joint conditioning:

C-UNIT-R
$$E\langle i\rangle \sim \mu \dashv\vdash \boldsymbol{C}_\mu\, v.\, \lceil E\langle i\rangle = v \rceil$$

C-FRAME
$$P * \boldsymbol{C}_\mu\, v.\, K(v) \vdash \boldsymbol{C}_\mu\, v.\, (P * K(v))$$

C-CONS
$$\frac{\forall v.\, K_1(v) \vdash K_2(v)}{\boldsymbol{C}_\mu\, v.\, K_1(v) \vdash \boldsymbol{C}_\mu\, v.\, K_2(v)}$$

Rule c-unit-r can convert back and forth from ownership of an expression E at i distributed as μ, and the conditioning on μ that makes E look deterministic. Rule c-frame allows to bring inside conditioning any resource that is independent from it. Rule c-cons simply allows to apply entailments inside joint conditioning. We can use these laws to perform conditioning on $\mathtt{m}\langle 1\rangle$:

$$\begin{aligned}
&\mathsf{k}\langle 1\rangle \sim \mathsf{Ber}_{1/2} * \mathtt{m}\langle 1\rangle \sim \mathsf{Ber}_p * \mathsf{c}\langle 2\rangle \sim \mathsf{Ber}_{1/2} \\
\vdash\ &\mathsf{k}\langle 1\rangle \sim \mathsf{Ber}_{1/2} * (\boldsymbol{C}_{\mathsf{Ber}_p}\, v.\, \lceil \mathtt{m}\langle 1\rangle = v \rceil) * \mathsf{c}\langle 2\rangle \sim \mathsf{Ber}_{1/2} && \text{(c-unit-r)}\\
\vdash\ &\boldsymbol{C}_{\mathsf{Ber}_p}\, v.\, (\lceil \mathtt{m}\langle 1\rangle = v \rceil * \mathsf{k}\langle 1\rangle \sim \mathsf{Ber}_{1/2} * \mathsf{c}\langle 2\rangle \sim \mathsf{Ber}_{1/2}) && \text{(c-frame)}
\end{aligned}$$

Here we use c-unit-r to convert ownership of $\mathtt{m}\langle 1\rangle$ into its conditioned form. Then we can bring the other independent variables inside the conditioning with c-frame. This derivation follows in spirit the way in which Lilac introduces conditioning, thus inheriting its ergonomic elegance. Our rules however differ from Lilac's in both form and substance. First, Lilac's rule for introducing conditioning (called C-Indep), requires converting ownership of a variable into conditioning, and bringing some independent resources inside conditioning, as a single monolithic step. In Bluebell we accomplish this pattern as a combination of our c-unit-r and c-frame, which are independently useful. Specifically, c-unit-r is bidirectional, which makes it useful to recover unconditional facts from conditional ones. Furthermore, we recognize that c-unit-r is nothing but a reflection of the right unit law of the monadic structure of distributions (which we elaborate on in Section 4). This connection prompted us to provide rules that reflect the remaining monadic laws (left unit c-unit-l and associativity c-fuse). It is noteworthy that these rules do not follow from Lilac's proofs: our modality has a different semantics, and our rules seamlessly apply to assertions of any arity.

To establish the conditional relational liftings of the entailment in (5), Bluebell provides a way to introduce relational liftings from ownership of the distributions of some variables:

$$\textsc{coupling}\quad \frac{\mu \circ \pi_1^{-1} = \mu_1 \qquad \mu \circ \pi_2^{-1} = \mu_2 \qquad \mu(R) = 1}{\mathsf{x}_1\langle 1\rangle \sim \mu_1 * \mathsf{x}_2\langle 2\rangle \sim \mu_2 \vdash \lfloor R(\mathsf{x}_1\langle 1\rangle, \mathsf{x}_2\langle 2\rangle)\rfloor}$$

The side conditions of the rule ask the prover to provide a coupling $\mu : \mathbb{D}(\mathbb{V} \times \mathbb{V})$ of $\mu_1 : \mathbb{D}(\mathbb{V})$ and $\mu_2 : \mathbb{D}(\mathbb{V})$, which assigns probability 1 to a (binary) relation R. If $\mathsf{x}_1\langle 1\rangle$ and $\mathsf{x}_2\langle 2\rangle$ are distributed as μ_1 and μ_2, respectively, then the relational lifting of R holds between them (as witnessed by the existence of μ). Note that for the rule to apply, the two variables need to live in distinct indices.

Interestingly, coupling can be derived from the encoding of relational lifting and the laws of joint conditioning.

Remarkably, although the rule mirrors the step of coupling two samplings in a pRHL proof, it does not apply to the code doing the sampling itself, but to the assertions representing the effects of those samplings. This allows us to delay the forming of coupling to until all necessary information is available (here, the outcome of $\mathtt{m}\langle 1\rangle$). We can use coupling to prove both entailments:

$$\mathsf{k}\langle 1\rangle \sim \mathsf{Ber}_{1/2} * \mathsf{c}\langle 2\rangle \sim \mathsf{Ber}_{1/2} \vdash \lfloor \mathsf{k}\langle 1\rangle = \mathsf{c}\langle 2\rangle \rfloor \quad\text{and}\quad \mathsf{k}\langle 1\rangle \sim \mathsf{Ber}_{1/2} * \mathsf{c}\langle 2\rangle \sim \mathsf{Ber}_{1/2} \vdash \lfloor \mathsf{k}\langle 1\rangle = \neg \mathsf{c}\langle 2\rangle \rfloor \tag{6}$$

In the first case we use the coupling which flips a single coin and returns the same outcome for both components, in the second we flip a single coin but return opposite outcomes. Thus we can now prove:

$$\boldsymbol{C}_{\mathsf{Ber}_p}\, v.\left(\lceil \mathtt{m}\langle 1\rangle = v \rceil * \left(\begin{matrix}\mathsf{k}\langle 1\rangle \sim \mathsf{Ber}_{1/2}\\ * \,\mathsf{c}\langle 2\rangle \sim \mathsf{Ber}_{1/2}\end{matrix}\right)\right) \vdash \boldsymbol{C}_{\mathsf{Ber}_p}\, v.\left(\lceil \mathtt{m}\langle 1\rangle = v \rceil * \begin{cases}\lfloor \mathsf{k}\langle 1\rangle = \mathsf{c}\langle 2\rangle \rfloor & \text{if } v = 0\\ \lfloor \mathsf{k}\langle 1\rangle = \neg\mathsf{c}\langle 2\rangle \rfloor & \text{if } v = 1\end{cases}\right)$$

by using c-cons, and using the two couplings of (6) in the $v = 0$ and $v = 1$ respectively. Finally, the assignment to `c` in `encrypt` generates the fact $\lceil \mathtt{c}\langle 1\rangle = \mathtt{k}\langle 1\rangle \mathbin{\mathtt{xor}} \mathtt{m}\langle 1\rangle \rceil$. By routine propagation of this fact we can establish $\boldsymbol{C}_{\mathrm{Ber}_p}\, v.\, \lfloor \mathtt{c}\langle 1\rangle = \mathtt{c}\langle 2\rangle \rfloor$. To get an unconditional lifting, we need a principle explaining the interaction between lifting and conditioning. Bluebell can derive the general rule:

rl-convex

$$\boldsymbol{C}_{\mu}\, \_.\, \lfloor R \rfloor \vdash \lfloor R \rfloor$$

which states that relational liftings are *convex*, i.e. closed under convex combinations.

> rl-convex is an instance of many rules on the interaction between relational lifting and the other connectives (conditioning in this case) that can be derived in Bluebell by exploiting the encoding of liftings as joint conditioning.

Let us see how this is done for rl-convex based on two other rules of joint conditioning:

c-skolem

$$\frac{\mu : \mathbb{D}(\Sigma_A)}{\boldsymbol{C}_{\mu}\, v.\, \exists x : X.\, Q(v, x) \vdash \exists f : A \to X.\, \boldsymbol{C}_{\mu}\, v.\, Q(v, f(v))}$$

c-fuse

$$\boldsymbol{C}_{\mu}\, v.\, \boldsymbol{C}_{\kappa(v)}\, w.\, K(v, w) \dashv\vdash \boldsymbol{C}_{\mu \prec \kappa}(v, w).\, K(v, w)$$

Rule c-skolem is a primitive rule which follows from Skolemization of the implicit universal quantification used on v by the modality. Rule c-fuse can be seen as a way to merge two nested conditioning or split one conditioning into two. The rule uses the operation $\mu \prec \kappa \triangleq \boldsymbol{\lambda}(v, w).\, \mu(v) \cdot \kappa(v)(w)$, a variant of **bind** that does not forget the intermediate v. Rule c-fuse is an immediate consequence of two primitive rules that reflect the associativity of the **bind** operation.

To prove rl-convex we start by unfolding the definition of relational lifting (we write $K(v)$ for the part of the encoding inside the conditioning):

$$\begin{aligned}
\boldsymbol{C}_{\mu}\, v.\, \lfloor R \rfloor \dashv\vdash\ & \boldsymbol{C}_{\mu}\, v.\, \exists \hat{\mu}_0.\, \boldsymbol{C}_{\hat{\mu}_0}\, w.\, K(w) & \\
\vdash\ & \exists \kappa.\, \boldsymbol{C}_{\mu}\, v.\, \boldsymbol{C}_{\kappa(v)}\, w.\, K(w) & (\text{c-skolem}) \\
\vdash\ & \exists \kappa.\, \boldsymbol{C}_{\mu \prec \kappa}(v, w).\, K(w) & (\text{c-fuse}) \\
\vdash\ & \exists \hat{\mu}_1.\, \boldsymbol{C}_{\hat{\mu}_1}(v, w).\, K(w) \vdash \lfloor R \rfloor & (\text{By def.})
\end{aligned}$$

The application of c-skolem commutes the existential quantification of the joint distribution $\hat{\mu}_0$ and the outer modality. By c-fuse we are able to merge the two modalities and obtain again something of the same form as the encoding of relational liftings.

2.4 Outside the Box of Relational Lifting

One of the well-known limitations of pRHL is that it requires a very strict structural alignment between the order of samplings to be coupled in the two programs. A common pattern that pRHL rules cannot handle is showing that reversing the order of execution of two blocks of code does not affect the output distribution, e.g. running `x := Ber(½);y := Ber(2/3)` versus `y := Ber(2/3);x := Ber(½)`. In Bluebell, we can establish this pattern using a *derived* general rule:

seq-swap

$$\frac{\{P_1\}[1{:}\, t_1, 2{:}\, t_1']\{\lfloor R_1 \rfloor\} \qquad \{P_2\}[1{:}\, t_2, 2{:}\, t_2']\{\lfloor R_2 \rfloor\}}{\{P_1 * P_2\}[1{:}\, (t_1 \,;\, t_2), 2{:}\, (t_2' \,;\, t_1')]\{\lfloor R_1 \land R_2 \rfloor\}}$$

The rule assumes that the lifting of R_1 (resp. R_2) can be established by analyzing t_1 and t_1' (t_2 and t_2') side by side from precondition P_1 (P_2). The standard sequential rule of pRHL would force an alignment between the wrong pairs (t_1 with t_2', and t_2 with t_1'). Crucial to the soundness of the rule is the assumption (expressed by the precondition $P_1 * P_2$ in the conclusion) that P_1 and P_2 are probabilistically independent.[1] In contrast, because pRHL lacks the construct of independence, it simply cannot express such a rule.

> Bluebell's treatment of relational lifting enables the study of the interaction between lifting and independence, unlocking a novel solution for forfeiting strict structural similarities between components required by relational logics.

Two ingredients of Bluebell cooperate to prove seq-swap: the adoption of a *weakest precondition* (WP) formulation of triples (and associated rules) and a novel property of relational lifting. Let us start with WP. In Bluebell, a triple $\{P\}\ \boldsymbol{t}\ \{Q\}$ is actually encoded as the entailment $P \vdash \mathbf{wp}\,\boldsymbol{t}\ \{Q\}$. Here, P and Q are both assertions on n-nary tuples of distributions; and throughout, we use the bold $\boldsymbol{t}$ to denote an n-nary tuple of program terms. The formula $\mathbf{wp}\,\boldsymbol{t}\ \{Q\}$ is a natural generalization of WP assertion to n-nary programs: $\mathbf{wp}\,\boldsymbol{t}\ \{Q\}$ holds on a n-nary tuple of distributions $\boldsymbol{\mu}$, if the tuple of output distributions obtained by running each program in $\boldsymbol{t}$ on the corresponding component of $\boldsymbol{\mu}$, satisfies Q. Bluebell provides a number of rules for manipulating WP; here is a selection needed for deriving seq-swap:

wp-cons

$$\frac{Q \vdash Q'}{\mathbf{wp}\,\boldsymbol{t}\ \{Q\} \vdash \mathbf{wp}\,\boldsymbol{t}\ \{Q'\}}$$

wp-frame

$$P * \mathbf{wp}\,\boldsymbol{t}\ \{Q\} \vdash \mathbf{wp}\,\boldsymbol{t}\ \{P * Q\}$$

wp-seq

$$\mathbf{wp}\ [i\colon t]\ \big\{\mathbf{wp}\ [i\colon t']\ \{Q\}\big\} \vdash \mathbf{wp}\ [i\colon (t;\ t')]\ \{Q\}$$

wp-nest

$$\mathbf{wp}\,\boldsymbol{t}_1\ \{\mathbf{wp}\,\boldsymbol{t}_2\ \{Q\}\} \dashv\vdash \mathbf{wp}\,(\boldsymbol{t}_1 \cdot \boldsymbol{t}_2)\ \{Q\}$$

Rules wp-cons and wp-frame are the usual consequence and framing rules of Separation Logic, in WP form. By adopting Lilac's measure-theoretic notion of independence as the interpretation for separating conjunction, we obtain a clean frame rule. Among the WP rules for program constructs, rule wp-seq takes care of sequential composition. Notably, we only need to state it for unary WPs, in contrast to other logics where supporting relational proofs requires building the lockstep strategy into the rules. We use the more flexible approach from the Logic for Hypertriple Composition (LHC) [D'Osualdo et al. 2022], where a handful of arity-changing rules allow seamless integration of unary and relational judgments. One such rule is the wp-nest rule, which establishes the equivalence of a WP on $\boldsymbol{t}_1 \cdot \boldsymbol{t}_2$, where $(\cdot)$ is union of indexed tuples with disjoint indexes, and two nested WPs involving $\boldsymbol{t}_1$, and $\boldsymbol{t}_2$ individually. This for instance allows us to lift the unary wp-seq to a binary lockstep rule:

$$\dfrac{\dfrac{\dfrac{\dfrac{P \vdash \mathbf{wp}\ [1\colon t_1]\ \{\mathbf{wp}\ [2\colon t_2]\ \{Q'\}\} \qquad Q' \vdash \mathbf{wp}\ [1\colon t_1']\ \{\mathbf{wp}\ [2\colon t_2']\ \{Q\}\}}{P \vdash \mathbf{wp}\ [1\colon t_1]\ \{\mathbf{wp}\ [2\colon t_2]\ \{\mathbf{wp}\ [1\colon t_1']\ \{\mathbf{wp}\ [2\colon t_2']\ \{Q\}\}\}\}}\ \textsc{wp-cons}}{P \vdash \mathbf{wp}\ [1\colon t_1]\ \{\mathbf{wp}\ [1\colon t_1']\ \{\mathbf{wp}\ [2\colon t_2]\ \{\mathbf{wp}\ [2\colon t_2']\ \{Q\}\}\}\}}\ \textsc{wp-nest}}{P \vdash \mathbf{wp}\ [1\colon (t_1\,;t_1')]\ \{\mathbf{wp}\ [2\colon (t_2\,;t_2')]\ \{Q\}\}}\ \textsc{wp-seq,wp-cons}}{P \vdash \mathbf{wp}\ [1\colon (t_1\,;t_1'),\, 2\colon (t_2\,;t_2')]\ \{Q\}}\ \textsc{wp-nest}$$

The crucial idea behind seq-swap is that the two programs t_1 and t_2 we want to swap rely on *independent* resources, and thus their effects are independent from each other. In Separation Logic this kind of reasoning is driven by framing: which is done through framing in Separation Logic:

[1] In the full model of Bluebell, to ensure safe mutation, assertions also include "write/read permissions" on variables (in the "variables as resource"-style [Bornat et al. 2005]). In seq-swap the separation between P_1 and P_2 ensures, in addition to probabilistic independence, that if t_1 has write permissions on a variable x, t_2 does not have read permissions on it and viceversa (and analogously for t_1' and t_2'). The full model incurs in some permissions bookkeeping, which we omit in this section for readability; Example 4.15 shows how to fill in the omitted details.

while executing t_1, frame the resources needed for t_2, which remain intact in the state left by t_1. Multiple applications of wp-frame and other basic rules get us to the post-condition $\lfloor R_1 \rfloor * \lfloor R_2 \rfloor$, but we want to combine them into one relational lifting. This is accommodated by:

$$\begin{array}{l}\textsc{rl-merge}\\ \lfloor R_1 \rfloor * \lfloor R_2 \rfloor \vdash \lfloor R_1 \land R_2 \rfloor\end{array}$$

We do not show the derivation here for space constraints, but essentially it consists in unfolding the encoding of lifting, and using c-frame and c-fuse to merge the two joint conditioning modalities.

Using these rules we can construct the following derivation:

$$\begin{array}{cl}
P_1 \vdash \mathbf{wp}\,[1\colon t_1, 2\colon t_1'] \, \{\lfloor R_1 \rfloor\} \qquad P_2 \vdash \mathbf{wp}\,[1\colon t_2, 2\colon t_2'] \, \{\lfloor R_2 \rfloor\} & \\ \hline
P_1 * P_2 \vdash \mathbf{wp}\,[1\colon t_1, 2\colon t_1'] \, \{\lfloor R_1 \rfloor\} * \mathbf{wp}\,[1\colon t_2, 2\colon t_2'] \, \{\lfloor R_2 \rfloor\} & \\ \hline
P_1 * P_2 \vdash \mathbf{wp}\,[1\colon t_1] \, \big\{\mathbf{wp}\,[1\colon t_2, 2\colon t_2'] \, \{\lfloor R_2 \rfloor\} * \mathbf{wp}\,[2\colon t_1'] \, \{\lfloor R_1 \rfloor\}\big\} & \textsc{wp-frame,wp-nest} \\ \hline
P_1 * P_2 \vdash \mathbf{wp}\,[1\colon t_1] \, \Big\{\mathbf{wp}\,[1\colon t_2, 2\colon t_2'] \, \big\{\lfloor R_2 \rfloor * \mathbf{wp}\,[2\colon t_1'] \, \{\lfloor R_1 \rfloor\}\big\}\Big\} & \textsc{wp-frame} \\ \hline
P_1 * P_2 \vdash \mathbf{wp}\,[1\colon t_1] \, \Big\{\mathbf{wp}\,[1\colon t_2, 2\colon t_2'] \, \big\{\mathbf{wp}\,[2\colon t_1'] \, \{\lfloor R_1 \rfloor * \lfloor R_2 \rfloor\}\big\}\Big\} & \textsc{wp-frame} \\ \hline
P_1 * P_2 \vdash \mathbf{wp}\,[1\colon (t_1\,;t_2)] \, \big\{\mathbf{wp}\,[2\colon (t_2'\,;t_1')] \, \{\lfloor R_1 \rfloor * \lfloor R_2 \rfloor\}\big\} & \textsc{wp-seq,wp-nest} \\ \hline
P_1 * P_2 \vdash \mathbf{wp}\,[1\colon (t_1\,;t_2), 2\colon (t_2'\,;t_1')] \, \big\{\lfloor R_1 \rfloor * \lfloor R_2 \rfloor\big\} & \textsc{wp-nest} \\ \hline
P_1 * P_2 \vdash \mathbf{wp}\,[1\colon (t_1\,;t_2), 2\colon (t_2'\,;t_1')] \, \{\lfloor R_1 \land R_2 \rfloor\} & \textsc{rl-merge}
\end{array}$$

We explain the proof strategy from bottom to top. We first apply rl-merge to the postcondition (thanks to wp-cons). This step reduces the goal to proving the two relational liftings can be established independently from each other. Then we apply wp-nest and wp-seq to separate the two indices, break the sequential compositions and recombine the two inner WPs. We then proceed by three applications of the wp-frame rule: the first brings $\lfloor R_2 \rfloor$ out of the innermost WP; the second brings the WP on $[1\colon t_1']$ outside the middle WP; the last brings the WP on $[1\colon t_2, 2\colon t_2']$ outside the topmost WP. An application of rule wp-nest merges the resulting nested WPs on t_1 and t_1'. We thus effectively reduced the problem to showing that the two WPs can be established independently, which was our original goal.

The rl-merge rule not only provides an elegant way of overcoming the long-standing alignments issue with constructing relational lifting, but also shows how fundamental the role of probabilistic independence is for compositional reasoning: the same rule with standard conjunction is unsound! Intuitively, if we just had $\lfloor R_1 \rfloor \land \lfloor R_2 \rfloor$, we would know there exist two couplings μ_1 and μ_2, justifying $\lfloor R_1 \rfloor$ and $\lfloor R_2 \rfloor$ respectively, but the desired consequence $\lfloor R_1 \land R_2 \rfloor$ requires the construction of a single coupling that justifies both relations at the same time. We can see this is not always possible by looking back at (6): for two fair coins we can establish $\lfloor \mathsf{k}\langle 1 \rangle = \mathsf{c}\langle 2 \rangle \rfloor \land \lfloor \mathsf{k}\langle 1 \rangle = \neg \mathsf{c}\langle 2 \rangle \rfloor$, but $\lfloor \mathsf{k}\langle 1 \rangle = \mathsf{c}\langle 2 \rangle \land \mathsf{k}\langle 1 \rangle = \neg \mathsf{c}\langle 2 \rangle \rfloor$ is equivalent to false.

3 PRELIMINARIES: PROGRAMS AND PROBABILITY SPACES

To formally define the model of Bluebell and validate its rules, we introduce a number of preliminary notions. Our starting point is the measure-theoretic approach of [Li et al. 2023a] in defining probabilistic separation. We recall the main definitions here. The main additional assumption we make throughout is that the set of outcomes of distributions is countable.

Definition 3.1 (Probability spaces). Given a set of possible *outcomes* Ω, a σ-algebra $\mathcal{F} \in \mathbb{A}(\Omega)$ is a set of subsets of Ω that is closed under countable unions and complements, and such that $\Omega \in \mathcal{F}$. The *full* σ-algebra over Ω is $\Sigma_\Omega = \wp(\Omega)$, the powerset of Ω. For $F \subseteq \wp(\Omega)$, we write $\sigma(F) \in \mathbb{A}(\Omega)$ for the smallest σ-algebra containing F. Given $\mathcal{F} \in \mathbb{A}(\Omega)$, a *probability distribution* $\mu \in \mathbb{D}(\mathcal{F})$, is a

$$\mathbb{E} \ni e ::= v \mid \mathsf{x} \mid \varphi(\vec{e}) \qquad \vec{e} ::= e_1,\dots,e_n \qquad \varphi ::= + \mid - \mid < \mid \dots \qquad d ::= \mathtt{Ber} \mid \mathtt{Unif} \mid \dots$$
$$\mathbb{T} \ni t ::= \mathsf{x} \mathrel{:=} e \mid \mathsf{x} \mathrel{:\approx} d(\vec{e}) \mid \textbf{skip} \mid t_1 ; t_2 \mid \textbf{if } e \textbf{ then } t_1 \textbf{ else } t_2 \mid \textbf{repeat } e\ t$$

Fig. 3. Syntax of program terms.

countably additive function $\mu\colon \mathcal{F} \to [0,1]$ with $\mu(\Omega) = 1$. The support of a distribution $\mu \in \mathbb{D}(\Sigma_\Omega)$ is the set of outcomes with non-zero probability $\operatorname{supp}(\mu) \triangleq \{a \in \Omega \mid \mu(a) > 0\}$, where $\mu(a)$ abbreviates $\mu(\{a\})$.

A *probability space* $\mathcal{P} \in \mathbb{P}(\Omega)$ is a pair $\mathcal{P} = (\mathcal{F}, \mu)$ of a σ-algebra $\mathcal{F} \in \mathbb{A}(\Omega)$ and a probability distribution $\mu \in \mathbb{D}(\mathcal{F})$. The *trivial* probability space $\mathbb{1}_\Omega \in \mathbb{P}(\Omega)$ is the trivial σ-algebra $\{\Omega, \emptyset\}$ equipped with the trivial probability distribution. Given $\mathcal{F}_1 \subseteq \mathcal{F}_2$ and $\mu \in \mathbb{D}(\mathcal{F}_2)$, the distribution $\mu|_{\mathcal{F}_1} \in \mathbb{D}(\mathcal{F}_1)$ is the restriction of μ to $\mathcal{F}_1$. The *extension pre-order* ($\sqsubseteq$) over probability spaces is defined as $(\mathcal{F}_1, \mu_1) \sqsubseteq (\mathcal{F}_2, \mu_2) \triangleq \mathcal{F}_1 \subseteq \mathcal{F}_2 \land \mu_1 = \mu_2|_{\mathcal{F}_1}$.

A function $f\colon \Omega_1 \to \Omega_2$ is *measurable on* $\mathcal{F}_1 \in \mathbb{A}(\Omega_1)$ *and* $\mathcal{F}_2 \in \mathbb{A}(\Omega_2)$ *if* $\forall X \in \mathcal{F}_2.\, f^{-1}(X) \in \mathcal{F}_1$. *When* $\mathcal{F}_2 = \Sigma_{\Omega_2}$ *we simply say* f *is measurable in* $\mathcal{F}_1$.

Definition 3.2 (Product and union spaces). Given $\mathcal{F}_1 \in \mathbb{A}(\Omega_1), \mathcal{F}_2 \in \mathbb{A}(\Omega_2)$, their product is the σ-algebra $\mathcal{F}_1 \otimes \mathcal{F}_2 \in \mathbb{A}(\Omega_1 \times \Omega_2)$ defined as $\mathcal{F}_1 \otimes \mathcal{F}_2 \triangleq \sigma(\{X_1 \times X_2 \mid X_1 \in \mathcal{F}_1, X_2 \in \mathcal{F}_2\})$, and their union is the σ-algebra $\mathcal{F}_1 \oplus \mathcal{F}_2 \triangleq \sigma(\mathcal{F}_1 \cup \mathcal{F}_2)$. The product of two probability distributions $\mu_1 \in \mathbb{D}(\mathcal{F}_1)$ and $\mu_2 \in \mathbb{D}(\mathcal{F}_2)$ is the distribution $(\mu_1 \otimes \mu_2) \in \mathbb{D}(\mathcal{F}_1 \otimes \mathcal{F}_2)$ defined by $(\mu_1 \otimes \mu_2)(X_1 \times X_2) = \mu_1(X_1)\mu_2(X_2)$ for all $X_1 \in \mathcal{F}_1, X_2 \in \mathcal{F}_2$.

Definition 3.3 (Independent product [Li et al. 2023a]). Given $(\mathcal{F}_1, \mu_1), (\mathcal{F}_2, \mu_2) \in \mathbb{P}(\Omega)$, their *independent product* is the probability space $(\mathcal{F}_1 \oplus \mathcal{F}_2, \mu) \in \mathbb{P}(\Omega)$ where for all $X_1 \in \mathcal{F}_1, X_2 \in \mathcal{F}_2$, $\mu(X_1 \cap X_2) = \mu_1(X_1) \cdot \mu_2(X_2)$. It is unique, if it exists [Li et al. 2023a, Lemma 2.3]. Let $\mathcal{P}_1 \circledast \mathcal{P}_2$ be the unique independent product of $\mathcal{P}_1$ and $\mathcal{P}_2$ when it exists, and be undefined otherwise.

Indexed tuples. To handle unary and higher-arity relational assertions in a uniform way, we consider finite sets of indices $I \subseteq \mathbb{N}$, and I-indexed tuples of values of type X, represented as (finite) functions X^I. We use boldface to range over such functions. The syntax $\boldsymbol{x} = [i_0{:}\, x_0, \dots, i_n{:}\, x_n]$ denotes the function $\boldsymbol{x} \in X^{\{i_0,\dots,i_n\}}$ with $\boldsymbol{x}(i_k) = x_k$. We often use comprehension-style notation e.g. $\boldsymbol{x} = [i{:}\, x_i \mid i \in I]$. For $\boldsymbol{x} \in A^I$ we let $|\boldsymbol{x}| \triangleq I$. Given some $\boldsymbol{x} \in A^I$ and some $J \subseteq I$, the operation $\boldsymbol{x} \setminus J \triangleq [i{:}\, \boldsymbol{x}(i) \mid i \in I \setminus J]$ removes the components with indices in J from $\boldsymbol{x}$.

Programs. We consider a simple first-order imperative language. We fix a finite set of *program variables* $\mathsf{x} \in \mathbb{X}$ and countable set of *values* $v \in \mathbb{V} \triangleq \mathbb{Z}$ and define the program *stores* to be $s \in \mathbb{S} \triangleq \mathbb{X} \to \mathbb{V}$ (note that $\mathbb{S}$ is countable).

Program *terms* $t \in \mathbb{T}$ are formed according to the grammar in Fig. 3. For simplicity, booleans are encoded by using $0 \in \mathbb{V}$ as false and any other value as true. We will use the events false $\triangleq \{0\}$ and true $\triangleq \{n \in \mathbb{V} \mid n \neq 0\}$. Programs use standard deterministic primitives φ, which are interpreted as functions $[\![\varphi]\!]\colon \mathbb{V}^n \to \mathbb{V}$, where n is the arity of φ. Expressions e are effect-free deterministic numeric expressions, and denote, as is standard, a function $[\![e]\!]\colon \mathbb{S} \to \mathbb{V}$, i.e. a random variable of $\Sigma_{\mathbb{S}}$. We write $\operatorname{pvar}(e)$ for the set of program variables that occur in e. Programs can refer to some collection of known *discrete* distributions d, each allowing a certain number of parameters. Sampling assignments $\mathsf{x} \mathrel{:\approx} d(\vec{v})$ sample from the distribution $[\![d]\!](\vec{v})\colon \mathbb{D}(\Sigma_{\mathbb{V}})$. The distribution $[\![\mathtt{Ber}]\!](p) = \mathtt{Ber}_p : \mathbb{D}(\Sigma_{\{0,1\}})$ is the Bernoulli distribution assigning probability p to outcome 1.

Similar to Lilac, we consider a simple iteration construct **repeat** $e\ t$ which evaluates e to a value $n \in \mathbb{V}$ and, if $n > 0$, executes t in sequence n times. This means we only consider a subset of terminating programs. The semantics of programs is entirely standard and is defined in [Bao et al. 2024]. It associates each term t to a function $[\![t]\!] \colon \mathbb{D}(\Sigma_{\mathbb{S}}) \longrightarrow \mathbb{D}(\Sigma_{\mathbb{S}})$ from distributions of input stores to distributions of output stores.

In the relational reasoning setting, one would consider multiple programs at the same time and relate their semantics. Following LHC [D'Osualdo et al. 2022], we define *hyper-terms* as $\boldsymbol{t} \in \mathbb{T}^J$ for some finite set of indices J. Let I be such that $|\boldsymbol{t}| \subseteq I$; the semantics $[\![\boldsymbol{t}]\!]_I \colon \mathbb{D}(\Sigma_{\mathbb{S}})^I \longrightarrow \mathbb{D}(\Sigma_{\mathbb{S}})^I$ takes a I-indexed family of distributions as input and outputs another I-indexed family of distributions:

$$[\![\boldsymbol{t}]\!]_I(\boldsymbol{\mu}) \triangleq \boldsymbol{\lambda} i.\, \mathsf{if}\ i \in |\boldsymbol{t}|\ \mathsf{then}\ [\![\boldsymbol{t}(i)]\!](\boldsymbol{\mu}(i))\ \mathsf{else}\ \boldsymbol{\mu}(i)$$

Note that the store distributions at indices in $I \setminus |t|$ are preserved as is. We omit I when it can be inferred from context. To refer to program variables in a specific component we use elements of $I \times \mathbb{X}$, writing $\mathsf{x}\langle i \rangle$ for (i, x).

4 THE BLUEBELL LOGIC

We are now ready to define Bluebell's semantic model, and formally prove its laws.

4.1 A Model of (Probabilistic) Resources

As a model for our assertions we use a modern presentation of partial commutative monoids, adapted from [Krebbers et al. 2018], called "ordered unital resource algebras" (henceforth RA). Instead of a partial binary operation, RAs are equipped with a *total* binary operation and a predicate $\mathcal{V}$ indicating which elements of the carrier are considered *valid* resources. Partiality of the operation manifests as mapping some combinations of arguments to invalid elements.

Definition 4.1 (Ordered Unital Resource Algebra). An *ordered unital resource algebra* (RA) is a tuple $(M, \preceq, \mathcal{V}, \cdot, \varepsilon)$ where $\preceq \colon M \times M \longrightarrow \mathrm{Prop}$ is the reflexive and transitive *resource order*, $\mathcal{V} \colon M \longrightarrow \mathrm{Prop}$ is the *validity predicate*, $(\cdot) \colon M \longrightarrow M \longrightarrow M$ is the *resource composition*, a commutative and associative binary operation on M, and $\varepsilon \in M$ is the *unit* of M, satisfying, for all $a, b, c \in M$:

$$\mathcal{V}(\varepsilon) \qquad\quad \varepsilon \cdot a = a \qquad\quad \frac{\mathcal{V}(a \cdot b)}{\mathcal{V}(a)} \qquad\quad \frac{a \preceq b \qquad \mathcal{V}(b)}{\mathcal{V}(a)} \qquad\quad \frac{a \preceq b}{a \cdot c \preceq b \cdot c}$$

Any RA can serve as a model of basic connectives of separation logics; in particular, $P * Q$ will hold on $a \in M$ if and only if there are $a_1, a_2 \in M$ such that $a = a_1 \cdot a_2$ and P holds on a_1 and Q holds on a_2.

Bluebell's assertions will be interpreted over a specific RA, which we construct as the combination of other basic RAs. The main component is the Probability Spaces RA, which uses independent product as the RA operation.

Definition 4.2 (Probability Spaces RA). The *probability spaces RA* PSp_Ω is the resource algebra $(\mathbb{P}(\Omega) \uplus \{\lightning\}, \preceq, \mathcal{V}, \cdot, \mathbb{1}_\Omega)$ where $\preceq$ is the extension order with $\lightning$ added as the top element, i.e. $\mathcal{P}_1 \preceq \mathcal{P}_2 \triangleq \mathcal{P}_1 \sqsubseteq \mathcal{P}_2$ and $\forall a \in \mathrm{PSp}_\Omega.\, a \preceq \lightning$; $\mathcal{V}(a) \triangleq a \neq \lightning$; composition is independent product:

$$a \cdot b \triangleq \begin{cases} \mathcal{P}_1 \circledast \mathcal{P}_2 & \text{if } a = \mathcal{P}_1, b = \mathcal{P}_2, \text{ and } \mathcal{P}_1 \circledast \mathcal{P}_2 \text{ is defined} \\ \lightning & \text{otherwise} \end{cases}$$

The fact that PSp_Ω satisfies the axioms of RAs is established in [Bao et al. 2024] and builds on the analogous construction in Lilac. In comparison with the coarser model of PSL, independent product represents a more sophisticated way of separating probability spaces. In PSL, separation of distributions requires the distributions to involve disjoint sets of variables, ruling out assertions like $\mathsf{x} \sim \mu * \lceil \mathsf{x} = \mathsf{y} \rceil$ or $\mathrm{own}(\mathsf{x}) * \mathrm{own}(\mathtt{x\ \ xor\ \ y})$, which are satisfiable in Lilac and Bluebell.

Example 4.3. Assume there are only two variables x and y. Let $X_v = \{s | s(\mathsf{x}) = v\}$ and $\mathcal{P}_1 = (\mathcal{F}_1, \mu_1)$ with $\mathcal{F}_1 = \sigma(\{X_v \mid v \in \mathbb{V}\})$ and let μ_1 give x the distribution of a fair coin, i.e. μ_1 is the extension to $\mathcal{F}_1$ of $\mu_1(X_0) = \mu_1(X_1) = 1/2$. Intuitively, the assertion $\mathsf{x} \sim \mathsf{Ber}_{1/2}$ holds on $\mathcal{P}_1$ (we will define the assertion's interpretation in Section 4.2). Similarly, $\lceil \mathsf{x} = \mathsf{y} \rceil$ holds on $\mathcal{P}_2 = (\mathcal{F}_2, \mu_2)$ where $\mathcal{F}_2 = \{\emptyset, \mathbb{S}, \{E\}, \mathbb{S} \setminus E\}$ with $E = \{s \mid s(\mathsf{x}) = s(\mathsf{y})\}$ and $\mu_2(E) = 1$. Note that $\mathcal{F}_2$ is very coarse: it does not contain events that can pin the value of x precisely; thanks to this, μ_2 does not need to specify what is the distribution of x, but only that y will coincide on x with probability 1. It is easy to see that the independent product of $\mathcal{P}_1$ and $\mathcal{P}_2$ exists and is $\mathcal{P}_3 = (\mathcal{F}_1 \oplus \mathcal{F}_2, \mu_3)$ where μ_3 is determined by $\mu_3(X_0 \cap E) = \mu_3(X_1 \cap E) = 1/2$, i.e. makes x y the outcomes of the same fair coin. This means $\mathcal{P}_3$ is a model of $\mathsf{x} \sim \mathsf{Ber}_{1/2} * \lceil \mathsf{x} = \mathsf{y} \rceil$.

When state is immutable, like in Lilac (which uses a functional language), the PSp_Ω RA is adequate to support a logic for probabilistic independence. There is however an obstacle in adopting independent product in a language with mutable state.

Example 4.4. Consider a simple assignment $\mathsf{x} \coloneqq \mathtt{0}$. In the spirit of separation logic's local reasoning, we would want to prove a small-footprint triple for the assignment, i.e. one where the precondition only involves ownership of the variable x. We could try with $x \sim \mu$ (for arbitrary μ) but we would run into problems proving the frame property: as we remarked, an assertion like $\lceil \mathsf{x} = \mathsf{y} \rceil$ is a valid frame of $x \sim \mu$; yet if $y \neq 0$, such frame would not hold after the assignment.

We solve this problem by combining PSp with an RA of permissions over variables. The idea is that in addition to information about the distribution, assertions can indicate which "write permissions" we own on variables. An assertion that owns write permissions on x would be incompatible with any frame predicating on x. Then a triple for assignment just needs to require write permission to the assigned variable. We model permissions using a standard fractional permission RA.

Definition 4.5. The *permissions RA* is defined as $(\mathsf{Perm}, \leq, \mathcal{V}, \cdot, \varepsilon)$ where $\mathsf{Perm} \triangleq \mathbb{X} \to \mathbb{Q}^+$, $a \leq b \triangleq \forall \mathsf{x} \in \mathbb{X}.\, a(\mathsf{x}) \leq b(\mathsf{x})$, $\mathcal{V}(a) \triangleq (\forall \mathsf{x} \in \mathbb{X}.\, a(\mathsf{x}) \leq 1)$, $a_1 \cdot a_2 \triangleq \boldsymbol{\lambda} \mathsf{x}.\, a_1(\mathsf{x}) + a_2(\mathsf{x})$ and $\varepsilon = \boldsymbol{\lambda}\_.\, 0$.

We now want to combine permissions with probability spaces. The goal is to allow probability spaces to contain only information about variables of which we have some non-zero permission. This gives rise to the following definition.

Definition 4.6 (Compatibility). Given a probability space $\mathcal{P} \in \mathbb{P}(\mathbb{S})$ and a permission map $p \in \mathsf{Perm}$, we say that $\mathcal{P}$ is *compatible* with p, written $\mathcal{P} \# p$, if there exists $\mathcal{P}' \in \mathbb{P}((\mathbb{X} \setminus S) \to \mathbb{V})$ such that $\mathcal{P} = \mathcal{P}' \otimes \mathbb{1}_{S \to \mathbb{V}}$, where $S = \{x \in \mathbb{X} \mid p(x) = 0\}$. Note that we are exploiting the isomorphism $\mathbb{S} \cong ((\mathbb{X} \setminus S) \to \mathbb{V}) \times (S \to \mathbb{V})$. We extend the notion to $\mathsf{PSp}_\mathbb{S}$ by declaring $\lightning \# p \triangleq \mathsf{True}$.

We can now construct an RA which combines probability spaces and permissions.

Definition 4.7. Let $\mathsf{PSpPm} \triangleq \{(\mathcal{P}_\lightning, p) \mid \mathcal{P}_\lightning \in \mathsf{PSp}_\mathbb{S}, p \in \mathsf{Perm}, \mathcal{P}_\lightning \# p\}$. We associate with PSpPm the *Probability Spaces with Permissions* RA $(\mathsf{PSpPm}, \leq, \mathcal{V}, \cdot, \varepsilon)$ where

$$\mathcal{V}((\mathcal{P}_\lightning, p)) \triangleq \mathcal{P}_\lightning \neq \lightning \land \forall \mathsf{x}. p(\mathsf{x}) \leq 1 \qquad (\mathcal{P}_\lightning, p) \cdot (\mathcal{P}'_\lightning, p') \triangleq (\mathcal{P}_\lightning \cdot \mathcal{P}'_\lightning, p \cdot p')$$
$$(\mathcal{P}_\lightning, p) \leq (\mathcal{P}'_\lightning, p') \triangleq \mathcal{P}_\lightning \leq \mathcal{P}'_\lightning \text{ and } p \leq p' \qquad \varepsilon \triangleq (\mathbb{1}_\mathbb{S}, \boldsymbol{\lambda} \mathsf{x}.\, 0)$$

Example 4.8. Using PSpPm, we can refine an assertion $\mathsf{x} \sim \mu$ into $(\mathsf{x} \sim \mu) @ (x{:}q)$, which holds on resources $(\mathcal{P}, p)$ where $\mathcal{P}$ distributes x as μ and $p(\mathsf{x}) \geq q$. What this achieves is to be able to differentiate between an assertion $(\mathsf{x} \sim \mu) @ (x{:}1/2)$ which allows frames to predicate on x (e.g. $\lceil \mathsf{x} = \mathsf{y} \rceil$) and an assertion $(\mathsf{x} \sim \mu) @ (x{:}1)$ which does not allow the frame and consequently allows mutation of x.

While this allows for a clean semantic treatment of mutation and independence, it does incur some bookkeeping of permissions in practice, which we omitted in the examples of Section 2. The necessary permissions are however easy to infer from the variables used in the assertions, as we will illustrate later in Example 4.15.

Finally, to build Bluebell's model we simply construct an RA of I-indexed tuples of probability spaces with permissions.

Definition 4.9 (Bluebell RA). Given a set of indices I and a RA M, the *product RA* M^I is the pointwise lifting of the components of M. Bluebell's model is $\mathcal{M}_I \triangleq \mathsf{PSpPm}^I$.

4.2 Probabilistic Hyper-Assertions

Now we turn to the assertions in our logic. We take a semantic approach to assertions: we do not insist on a specific syntax and instead characterize what constitutes an assertion by its type. In Separation Logic, assertions are defined relative to some RA M, as the upward closed functions $M \to \mathrm{Prop}$. An assertion $P\colon M \to \mathrm{Prop}$ is upward closed if $\forall a, a' \in M.\, a \preceq_M a' \Rightarrow P(a) \Rightarrow P(a')$. We write $M \xrightarrow{u} \mathrm{Prop}$ for the type of upward closed assertions on M. We define hyper-assertions to be assertions over $\mathcal{M}_I$, i.e. $P \in \mathsf{HA}_I \triangleq \mathcal{M}_I \xrightarrow{u} \mathrm{Prop}$. Entailment is defined as $(P \vdash Q) \triangleq \forall a \in M.\, \mathcal{V}(a) \Rightarrow (P(a) \Rightarrow Q(a))$. Logical equivalence is defined as entailment in both directions: $P \dashv\vdash Q \triangleq (P \vdash Q) \wedge (Q \vdash P)$. We inherit the basic connectives (conjunction, disjunction, separation, quantification) from SL, which are well-defined on arbitrary RAs, including $\mathcal{M}_I$. In particular:

$$P * Q \triangleq \boldsymbol{\lambda} a.\, \exists b_1, b_2.\, (b_1 \cdot b_2) \leq a \wedge P(b_1) \wedge Q(b_2) \qquad \ulcorner \phi \urcorner \triangleq \boldsymbol{\lambda}\_.\, \phi \qquad \mathsf{Own}(b) \triangleq \boldsymbol{\lambda} a.\, b \leq a$$

Pure assertions $\ulcorner \phi \urcorner$ lift meta-level propositions ϕ to assertions (by ignoring the resource). $\mathsf{Own}(b)$ holds on resources that are greater or equal than b in the RA order; this means b represents a lower bound on the available resources.

We now turn to assertions that are specific to probabilistic reasoning in Bluebell, i.e. the ones that can only be interpreted in $\mathcal{M}_I$. We use the following two abbreviations:

$$\mathsf{Own}(\boldsymbol{\mathcal{F}}, \boldsymbol{\mu}, \boldsymbol{p}) \triangleq \mathsf{Own}(((\boldsymbol{\mathcal{F}}, \boldsymbol{\mu}), \boldsymbol{p})) \qquad \mathsf{Own}(\boldsymbol{\mathcal{F}}, \boldsymbol{\mu}) \triangleq \exists \boldsymbol{p}.\, \mathsf{Own}(\boldsymbol{\mathcal{F}}, \boldsymbol{\mu}, \boldsymbol{p})$$

To start, we define *A-typed assertion expressions E* which are of type $E\colon \mathbb{S} \to A$. Note that the type of the semantics of a program expression $[\![e]\!]\colon \mathbb{S} \to \mathbb{V}$ is a $\mathbb{V}$-typed assertion expression; because of this we seamlessly use program expressions in assertions, implicitly coercing them to their semantics. Since in general we deal with hyper-stores $\boldsymbol{s} \in \mathbb{S}^I$, we use the notation $E\langle i \rangle$ to denote the application of E to the store $\boldsymbol{s}(i)$. Notationally, it may be confusing to read composite expressions like $(\mathsf{x} - \mathsf{z})\langle i \rangle$, so we write them for clarity with each program variable annotated with the index, as in $\mathsf{x}\langle i \rangle - \mathsf{z}\langle i \rangle$.

The meaning of owning $\mathsf{x}\langle 1 \rangle \sim \mu$. A function $f\colon A \to B$ is *measurable* in a σ-algebra $\mathcal{F} : \mathbb{A}(A)$ if $f^{-1}(b) = \{a \in A \mid f(a) = b\} \in \mathcal{F}$ for all $b \in B$. An expression E always defines a measurable function (i.e. a *random variable*) in $\Sigma_{\mathbb{S}}$, but might not be measurable in some sub-algebras of $\Sigma_{\mathbb{S}}$. Lilac proposed to use measurability as the notion of ownership: an expression E is owned in any resources that contains enough information to determine its distribution, i.e. that makes E measurable. While this makes sense conceptually, we discovered it made another important connective of Lilac, almost sure equality, slightly flawed (in that it would not support the necessary laws).[2] We propose a slight weakening of the notion of measurability which solves those issues while still retaining the intent behind the meaning of ownership in relation to independence and conditioning. We call this weaker notion "almost measurability".

[2] In fact, a later revision [Li et al. 2023b] corrected the issue, although with a different solution from ours. See Section 6.

Definition 4.10 (Almost-measurability). Given a probability space $(\mathcal{F}, \mu) \in \mathbb{P}(\Omega)$ and a set $X \subseteq \Omega$, we say X is *almost measurable* in $(\mathcal{F}, \mu)$, written $X \prec (\mathcal{F}, \mu)$, if

$$\exists X_1, X_2 \in \mathcal{F}.\, X_1 \subseteq X \subseteq X_2 \land \mu(X_1) = \mu(X_2)$$

We say a function $E\colon \Omega \to A$, is *almost measurable* in $(\mathcal{F}, \mu)$, written $E \prec (\mathcal{F}, \mu)$, if $E^{-1}(a) \prec (\mathcal{F}, \mu)$ for all $a \in A$. When $X_1 \subseteq X \subseteq X_2$ and $\mu(X_1) = \mu(X_2) = p$, we can unambiguously assign probability p to X, as any extension of μ to Σ_Ω must assign p to X; then we write $\mu(X)$ for p.

While almost-measurability does not imply measurability, it constrains the current probability space to contain enough information to uniquely determine the distribution of E in any extension where E becomes measurable. For example let $X = \{s \mid s(\mathsf{x}) = 42\}$ and $\mathcal{F} = \sigma(\{X\}) = \{\mathbb{S}, \emptyset, X, \mathbb{S} \setminus X\}$. If $\mu(X) = 1$, then $\mathsf{x} \prec (\mathcal{F}, \mu)$ holds but x is not measurable in $\mathcal{F}$, as $\mathcal{F}$ lacks events for $\mathsf{x} = v$ for all v except 42. Nevertheless, any extension $(\mathcal{F}', \mu') \sqsupseteq (\mathcal{F}, \mu)$ where x is measurable, would need to assign $\mu'(X) = 1$ and $\mu(\mathsf{x} = v) = 0$ for every $v \neq 42$.

We arrive at the definition of the assertion $E\langle i\rangle \sim \mu$ which requires $E\langle i\rangle$ to be almost-measurable, determining its distribution as μ in any extension of the local probability space. Formally, given $\mu : \mathbb{D}(\Sigma_A)$ and $E\colon \mathbb{S} \to A$, we define:

$$E\langle i\rangle \sim \mu \triangleq \exists \boldsymbol{\mathcal{F}}, \boldsymbol{\mu}.\, \mathsf{Own}(\boldsymbol{\mathcal{F}}, \boldsymbol{\mu}) * \ulcorner E \prec (\boldsymbol{\mathcal{F}}(i), \boldsymbol{\mu}(i)) \land \mu = \boldsymbol{\mu}(i) \circ E^{-1} \urcorner$$

The assertion states that we own just enough information about the probability space at index i, so that its distribution is uniquely determined as μ in any extension of the space.

Using the $E\langle i\rangle \sim \mu$ assertion we can define a number of useful derived assertions:

$$\mathsf{E}[E\langle i\rangle] = r \triangleq \exists \mu.\, E\langle i\rangle \sim \mu * \ulcorner r = \textstyle\sum_{a \in \mathrm{supp}(\mu)} \mu(a) \cdot a \urcorner \qquad \lceil E\langle i\rangle \rceil \triangleq (E \in \mathsf{true})\langle i\rangle \sim \delta_{\mathsf{True}}$$

$$\mathsf{Pr}(E\langle i\rangle) = r \triangleq \exists \mu.\, E\langle i\rangle \sim \mu * \ulcorner \mu(\mathsf{true}) = r \urcorner \qquad \mathsf{own}(E\langle i\rangle) \triangleq \exists \mu.\, E\langle i\rangle \sim \mu$$

Assertions about expectations ($\mathsf{E}[E\langle i\rangle]$) and probabilities ($\mathsf{Pr}(E\langle i\rangle)$), simply assert ownership of some distribution with the desired (pure) property. The "almost surely" assertion $\lceil E\langle i\rangle \rceil$ takes a boolean-valued expression E and asserts that it holds (at i) with probability 1. As remarked in Example 4.3, an assertion like $\lceil \mathsf{x}\langle 1\rangle = \mathsf{y}\langle 1\rangle \rceil$ owns the expression $(\mathsf{x}\langle 1\rangle = \mathsf{y}\langle 1\rangle)$ but not necessarily $\mathsf{x}\langle 1\rangle$ itself: the only events needed to make the expression almost measurable are $\mathsf{x}\langle 1\rangle = \mathsf{y}\langle 1\rangle$ and $\mathsf{x}\langle 1\rangle \neq \mathsf{y}\langle 1\rangle$, which would be not enough to make $\mathsf{x}\langle 1\rangle$ itself almost measurable. This means that an assertion like $\mathsf{own}(\mathsf{x}\langle 1\rangle) * \lceil \mathsf{x}\langle 1\rangle = \mathsf{y}\langle 1\rangle \rceil$ is satisfiable.

Permissions. The previous example highlights the difficulty with supporting mutable state: owning $\mathsf{x}\langle 1\rangle \sim \mu$ is not enough to allow safe mutation, because the frame can record information like $\lceil \mathsf{x}\langle 1\rangle = \mathsf{y}\langle 1\rangle \rceil$, which could be invalidated by an assignment to x. Our solution is analogous to the "variables as resource" technique in Separation Logic [Bornat et al. 2005], and uses the permission component of Bluebell's RA. To manipulate permissions we define the assertions:

$$(\mathsf{x}\langle i\rangle{:}q) \triangleq \exists \boldsymbol{\mathcal{P}}, \boldsymbol{p}.\, \mathsf{Own}(\boldsymbol{\mathcal{P}}, \boldsymbol{p}) * \ulcorner \boldsymbol{p}(i)(\mathsf{x}) = q \urcorner \qquad P@\boldsymbol{p} \triangleq P \land \exists \boldsymbol{\mathcal{P}}.\, \mathsf{Own}(\boldsymbol{\mathcal{P}}, \boldsymbol{p})$$

Now owning $(\mathsf{x}\langle 1\rangle{:}1)$ forbids any frame to retain information about $\mathsf{x}\langle 1\rangle$: any resource compatible with $(\mathsf{x}\langle 1\rangle{:}1)$ would have a σ-algebra which is trivial on $\mathsf{x}\langle 1\rangle$. In practice, preconditions are always of the form $P@\boldsymbol{p}$ where $\boldsymbol{p}$ contains full permissions for every variable the relevant program mutates, and non-zero permissions for the other variables referenced in the assertions or program. When framing, one would distribute evenly the permissions to each separated conjunct, according to the variables mentioned in the assertions. We illustrate this pattern concretely in Example 4.15.

c-true

$$\vdash \mathbf{C}_{\mu}\, \_.\, \mathsf{True}$$

c-unit-l

$$\mathbf{C}_{\delta_{v_0}}\, v.\, K(v) \dashv\vdash K(v_0)$$

c-transf

$$\frac{\begin{array}{c} f\colon \operatorname{supp}(\mu') \to \operatorname{supp}(\mu) \ \text{ bijective} \\ \forall b \in \operatorname{supp}(\mu').\, \mu'(b) = \mu(f(b)) \end{array}}{\mathbf{C}_{\mu}\, a.\, K(a) \vdash \mathbf{C}_{\mu'}\, b.\, K(f(b))}$$

c-and

$$\frac{\operatorname{idx}(K_1) \cap \operatorname{idx}(K_2) = \emptyset}{\mathbf{C}_{\mu}\, v.\, K_1(v) \land \mathbf{C}_{\mu}\, v.\, K_2(v) \vdash \mathbf{C}_{\mu}\, v.\, (K_1(v) \land K_2(v))}$$

sure-str-convex

$$\mathbf{C}_{\mu}\, v.\, (K(v) * \lceil E\langle i\rangle \rceil) \vdash \lceil E\langle i\rangle \rceil * \mathbf{C}_{\mu}\, v.\, K(v)$$

c-pure

$$\ulcorner \mu(X) = 1 \urcorner * \mathbf{C}_{\mu}\, v.\, K(v) \dashv\vdash \mathbf{C}_{\mu}\, v.\, (\ulcorner v \in X \urcorner * K(v))$$

Fig. 4. Primitive Conditioning Laws.

Relevant indices. Sometimes it is useful to determine which indices are relevant for an assertion. Semantically, we can determine if the indices $J \subseteq I$ are irrelevant to P by $\operatorname{irrel}_J(P) \triangleq \forall a \in \mathcal{M}_I.\, \big(\exists a'.\, \mathcal{V}(a') \land a = a' \setminus J \land P(a')\big) \implies P(a)$. The set $\operatorname{idx}(P)$ is the smallest subset of I so that $\operatorname{irrel}_{I \setminus \operatorname{idx}(P)}(P)$ holds. Rule and-to-star states that separation between resources that live in different indexes is the same as normal conjunction: distributions at different indexes are neither independent nor correlated; they simply live in "parallel universes" and can be related as needed.

and-to-star

$$\frac{\operatorname{idx}(P) \cap \operatorname{idx}(Q) = \emptyset}{P \land Q \vdash P * Q}$$

4.3 Joint Conditioning

As we discussed in Section 2, the centerpiece of Bluebell is the joint conditioning modality, which we can now define fully formally.

Definition 4.11 (Joint conditioning modality). Let $\mu \in \mathbb{D}(\Sigma_A)$ and $K\colon A \to \mathsf{HA}_I$, then we define the assertion $\mathbf{C}_{\mu}\, K : \mathsf{HA}_I$ as follows (where $\boldsymbol{\kappa}(I)(v) \triangleq [i\colon \boldsymbol{\kappa}(i)(v) \mid i \in I]$):

$$\begin{aligned} \mathbf{C}_{\mu}\, K \triangleq \boldsymbol{\lambda} a.\, \exists \boldsymbol{\mathcal{F}}, \boldsymbol{\mu}, \boldsymbol{p}, \boldsymbol{\kappa}.\, (\boldsymbol{\mathcal{F}}, \boldsymbol{\mu}, \boldsymbol{p}) \leq a \land \forall i \in I.\, \boldsymbol{\mu}(i) = \mathbf{bind}(\mu, \boldsymbol{\kappa}(i)) \\ \land \forall v \in \operatorname{supp}(\mu). K(v)(\boldsymbol{\mathcal{F}}, \boldsymbol{\kappa}(I)(v), \boldsymbol{p}) \end{aligned}$$

The definition follows the principle we explained in Section 2.2: $\mathbf{C}_{\mu}\, K$ holds on resources where we own some tuple of probability spaces which can all be seen as the convex combinations of the same μ and some kernel. Then the conditional assertion $K(v)$ is required to hold on the tuple of kernels evaluated at v. Note that the definition is upward-closed by construction.

We discussed a number of joint conditioning laws in Section 2. Figure 4 shows some important primitive laws that were left out. Rule c-true allows to introduce a trivial modality; together with c-frame this allows for the introduction of the modality around any assertion. Rule c-unit-l is a reflection of the left unit rule of the underlying monad: conditioning on the Dirac distribution can be eliminated. Rule c-transf allows for the transformation of the convex combination using μ into using μ' by applying a bijection between their support in a way that does not affect the weights of each outcome. Rule c-and allows to merge two modalities using the same μ, provided the inner conditioned assertions do not overlap in their relevant indices.

Rule sure-str-convex internalizes a stronger version of convexity of $\lceil E\langle i\rangle \rceil$ assertions. When $K(v) = \mathsf{True}$ we obtain convexity $\mathbf{C}_{\mu}\, v.\, \lceil E\langle i\rangle \rceil \vdash \lceil E\langle i\rangle \rceil$. Additionally the rule asserts that the unconditional $\lceil E\langle i\rangle \rceil$ keeps being independent of the conditional K.

Finally, rule c-pure allows to translate facts that hold with probability 1 in μ to predicates that hold on every v bound by conditioning on μ.

We can now give the general encoding of relational lifting in terms of joint conditioning.

Definition 4.12 (Relational Lifting). Let $X \subseteq \mathbb{I} \times \mathbb{X}$; given a relation R between variables in X, i.e. $R \subseteq \mathbb{V}^X$, we define (letting $\lceil \mathsf{x}\langle i\rangle = \boldsymbol{v}(\mathsf{x}\langle i\rangle)\rceil_{\mathsf{x}\langle i\rangle \in X} \triangleq \bigwedge_{\mathsf{x}\langle i\rangle \in X} \lceil \mathsf{x}\langle i\rangle = \boldsymbol{v}(\mathsf{x}\langle i\rangle)\rceil$):

$$\lfloor R \rfloor \triangleq \exists \mu : \mathbb{D}(\mathbb{V}^X).\ \ulcorner \mu(R) = 1 \urcorner * \boldsymbol{C}_\mu\, \boldsymbol{v}.\ \lceil \mathsf{x}\langle i\rangle = \boldsymbol{v}(\mathsf{x}\langle i\rangle)\rceil_{\mathsf{x}\langle i\rangle \in X}$$

Example 4.13. Let us expand Definition 4.12 on $\lfloor \mathsf{k}\langle 1\rangle = \mathsf{c}\langle 2\rangle \rfloor$. The assertion concerns the variables $X = \{\mathsf{k}\langle 1\rangle, \mathsf{c}\langle 2\rangle\}$; to instantiate the definition we see the assertion as the lifting $\lfloor R_= \rfloor$ of the relation $R_= \subseteq \mathbb{V}^X$ defined as $R_= = \{\boldsymbol{v} \in \mathbb{V}^X \mid \boldsymbol{v}(\mathsf{k}\langle 1\rangle) = \boldsymbol{v}(\mathsf{c}\langle 2\rangle)\}$, giving rise to the assertion

$$\exists \mu.\ \ulcorner \mu(R_=) = 1 \urcorner * \boldsymbol{C}_\mu\, \boldsymbol{v}.\ \lceil \mathsf{k}\langle 1\rangle = \boldsymbol{v}(\mathsf{k}\langle 1\rangle)\rceil \land \lceil \mathsf{c}\langle 2\rangle = \boldsymbol{v}(\mathsf{c}\langle 2\rangle)\rceil$$

Here, $\mathbb{V}^X$ can be alternatively presented as $\mathbb{V}^2$, giving $R_= \equiv \{(v, v) \mid v \in \mathbb{V}\}$. With this reformulation, the encoding of Definition 4.12 becomes

$$\exists \mu.\ \ulcorner \mu(R_=) = 1 \urcorner * \boldsymbol{C}_\mu (v_1, v_2).\ \lceil \mathsf{k}\langle 1\rangle = v_1 \rceil \land \lceil \mathsf{c}\langle 2\rangle = v_2 \rceil$$

Thanks to C-PURE, the assertion can be rewritten as $\exists \mu.\ \boldsymbol{C}_\mu (v_1, v_2).\ \ulcorner R_=(v_1, v_2) \urcorner * \lceil \mathsf{k}\langle 1\rangle = v_1 \rceil \land \lceil \mathsf{c}\langle 2\rangle = v_2 \rceil$ which can be simplified to $\exists \mu.\ \boldsymbol{C}_\mu (v_1, v_2).\ \big(\lceil \mathsf{k}\langle 1\rangle = v_1 \rceil \land \lceil \mathsf{c}\langle 2\rangle = v_2 \rceil \land \ulcorner v_1 = v_2 \urcorner\big)$ (which is how we presented the encoding in Section 2.2). Since $R_=$ is so simple, by C-TRANSF we can simplify the assertion even further and obtain $\exists \mu.\ \boldsymbol{C}_\mu\, v.\ \big(\lceil \mathsf{k}\langle 1\rangle = v \rceil \land \lceil \mathsf{c}\langle 2\rangle = v \rceil\big)$.

In rule RL-MERGE, the two relations might refer to different indexed variables, i.e. $R_1 \in \mathbb{V}^{X_1}$ and $R_2 \in \mathbb{V}^{X_2}$; the notation $R_1 \land R_2$ is defined as $R_1 \land R_2 \triangleq \big\{ \boldsymbol{s} \in \mathbb{V}^{X_1 \cup X_2} \,\big|\, \boldsymbol{s}|_{X_1} \in R_1 \land \boldsymbol{s}|_{X_2} \in R_2 \big\}$.

4.4 Weakest Precondition

To reason about (hyper-)programs, we introduce a *weakest-precondition assertion* (WP) $\mathbf{wp}\, \boldsymbol{t}\, \{Q\}$, which intuitively states: given the current input distributions (at each index), if we run the programs in $\boldsymbol{t}$ at their corresponding index we obtain output distributions that satisfy Q; furthermore, every frame is preserved. We refer to the number of indices of $\boldsymbol{t}$ as the *arity* of the WP.

Definition 4.14 (Weakest Precondition). For $a \in \mathcal{M}_I$ and $\boldsymbol{\mu} : \mathbb{D}(\Sigma_{\mathbb{S}^I})$ let $a \leq \boldsymbol{\mu}$ mean $a \leq (\Sigma_{\mathbb{S}^I}, \boldsymbol{\mu}, \boldsymbol{\lambda} x.\, 1)$.

$$\mathbf{wp}\, \boldsymbol{t}\, \{Q\} \triangleq \boldsymbol{\lambda} a.\, \forall \boldsymbol{\mu}_0. \forall c.\, (a \cdot c) \leq \boldsymbol{\mu}_0 \Rightarrow \exists b.\, \big((b \cdot c) \leq [\![\boldsymbol{t}]\!](\boldsymbol{\mu}_0) \land Q(b)\big)$$

The assertion holds on the resources a such that if, together with some frame c, they can be seen as a fragment of the global distribution $\boldsymbol{\mu}_0$, then it is possible to update the resource to some b which still composes with the frame c, and $b \cdot c$ can be seen as a fragment of the output distribution $[\![\boldsymbol{t}]\!](\boldsymbol{\mu}_0)$. Moreover, such b needs to satisfy the postcondition Q.

We discussed some of the WP rules of BLUEBELL in Section 2; the full set of rules is produced in [Bao et al. 2024]. Let us briefly mention the axioms for assignments:

WP-SAMP

$$(\mathsf{x}\langle i\rangle{:}1) \vdash \mathbf{wp}\, [i{:}\, \mathsf{x} \mathrel{:\approx} d(\vec{v})]\, \{\mathsf{x}\langle i\rangle \sim d(\vec{v})\}$$

WP-ASSIGN

$$\frac{\mathsf{x} \notin \operatorname{pvar}(e) \qquad \forall \mathsf{y} \in \operatorname{pvar}(e).\, \boldsymbol{p}(\mathsf{y}\langle i\rangle) > 0 \qquad \boldsymbol{p}(\mathsf{x}\langle i\rangle) = 1}{(\boldsymbol{p}) \vdash \mathbf{wp}\, [i{:}\, \mathsf{x} := e]\, \big\{\lceil \mathsf{x}\langle i\rangle = e\langle i\rangle \rceil @ \boldsymbol{p}\big\}}$$

Rule WP-SAMP is the expected "small footprint" rule for sampling; the precondition only requires full permission on the variable being assigned, to forbid any frame to record information about it. Rule WP-ASSIGN requires full permission on x, and non-zero permission on the variables on the RHS of the assignment. This allows the postcondition to assert that x and the expression e assigned to it are equal with probability 1. The condition $\mathsf{x} \notin \operatorname{pvar}(\vec{e})$ ensures e has the same meaning before and after the assignment, but is not restrictive: if needed the old value of x can be stored in a temporary variable, or the proof can condition on x to work with its pure value.

The assignment rules are the only ones that impose constraints on the owned permissions. In proofs, this means that most manipulations simply thread through permissions so that the needed ones can reach the applications of the assignment rules. To avoid cluttering derivations with this bookkeeping, we mostly omit permission information from assertions. The appropriate permission annotations can be easily inferred, as we show in the following example.

Example 4.15. Consider the following triple with permissions omitted:

$$\mathsf{x}\langle 1\rangle \sim \mu_1 * \lceil \mathsf{x}\langle 1\rangle = \mathsf{y}\langle 1\rangle \rceil * \mathsf{z}\langle 1\rangle \sim \mu_2 \vdash \mathbf{wp}\,[1{:}\,\mathsf{x} := \mathsf{z}]\,\{\lceil \mathsf{x}\langle 1\rangle = \mathsf{z}\langle 1\rangle \rceil * \mathsf{z}\langle 1\rangle \sim \mu_2\}$$

To be able to apply rule wp-assign, we need to get $(x{:}1, z{:}q)$ from the precondition, for some $q > 0$. To do so, formally, we need to be more explicit about the permissions owned. The pattern is that whenever a triple is considered, the precondition should own full permissions on the variables assigned to in the program term, and non-zero permission on the other relevant variables. In our example we need permission 1 for $\mathsf{x}\langle 1\rangle$ and arbitrary permissions $q_1, q_2 > 0$ for $\mathsf{y}\langle 1\rangle$ and $\mathsf{z}\langle 1\rangle$ respectively. Since we have two separated sub-assertions that refer to $\mathsf{x}\langle 1\rangle$, we would split the full permission into two halves, obtaining the precondition:

$$(\mathsf{x}\langle 1\rangle \sim \mu_1)@(\mathsf{x}\langle 1\rangle{:}\tfrac{1}{2}) * \lceil \mathsf{x}\langle 1\rangle = \mathsf{y}\langle 1\rangle \rceil @(\mathsf{x}\langle 1\rangle{:}\tfrac{1}{2}, \mathsf{y}\langle 1\rangle{:}q_1) * (\mathsf{z}\langle 1\rangle \sim \mu_2)@(\mathsf{z}\langle 1\rangle{:}q_2)$$

To obtain the full permission on $\mathsf{x}\langle 1\rangle$ we are now forced to consume both the first two resources, weakening the precondition to:

$$(\mathsf{x}\langle 1\rangle{:}1) * (\mathsf{y}\langle 1\rangle{:}q_1) * (\mathsf{z}\langle 1\rangle \sim \mu_2)@(\mathsf{z}\langle 1\rangle{:}q_2)$$

This step in general forces the consumption of any frame recording information about the assigned variables. To obtain non-zero permission for $\mathsf{z}\langle 1\rangle$ while still being able to frame $\mathsf{z}\langle 1\rangle \sim \mu_2$, we let $q = q_2/2$ and weaken the precondition to:

$$(\mathsf{x}\langle 1\rangle{:}1, \mathsf{z}\langle 1\rangle{:}q) * (\mathsf{y}\langle 1\rangle{:}q_1) * (\mathsf{z}\langle 1\rangle \sim \mu_2)@(\mathsf{z}\langle 1\rangle{:}q)$$

Now an application of wp-frame and wp-assign would give us a postcondition:

$$\lceil \mathsf{x}\langle 1\rangle = \mathsf{z}\langle 1\rangle \rceil @(\mathsf{x}\langle 1\rangle{:}1, \mathsf{z}\langle 1\rangle{:}q) * (\mathsf{y}\langle 1\rangle{:}q_1) * (\mathsf{z}\langle 1\rangle \sim \mu_2)@(\mathsf{z}\langle 1\rangle{:}q)$$

which is strong enough to imply the desired postcondition. In a fully expanded proof, one would keep the permissions owned in the postcondition so that they can be used in proofs concerning the continuation of the program.

5 CASE STUDIES FOR BLUEBELL

Our evaluation of Bluebell is based on two main lines of enquiry: (1) Are high-level principles about probabilistic reasoning provable from the core constructs of Bluebell? (2) Does Bluebell, through enabling new reasoning patterns, expand the horizon for verification of probabilistic programs *beyond* what was possible before? We include case studies that try to highlight the contribution of Bluebell each question, and sometimes both at the same time. Specifically, our evaluation is guided by the following research questions:

RQ1: Do joint conditioning and independence offer a good abstract interface over the underlying semantic model?

RQ2: Can known unary/relational principles be reconstructed from Bluebell's primitives?

RQ3: Can new unary/relational principles be discovered (as new lemmas) and proved from Bluebell's primitives?

RQ4: Can Bluebell's primitives be successfully incorporated in an effective *program* logic?

We already demonstrated positive answers to some of these questions in Section 2: for example, the proof of the One-time pad addresses **RQ1** and **RQ2**, the proof of SEQ-SWAP addresses **RQ3** and **RQ4**. In this section we provide a more detailed evaluation through a number of challenging examples. The full proofs of the case studies and additional examples are in [Bao et al. 2024]. Here, we summarize some highlights to frame the key contributions of Bluebell.

5.1 pRHL-style Reasoning

Our first example is an encoding of pRHL's judgments in Bluebell, sketching how pRHL-style reasoning can be effectively embedded and extended in Bluebell (**RQ1** to **RQ4**).

In pRHL, the semantics of triples implicitly always conditions on the input store, so that programs are always seen as running from a pair of *deterministic* input stores satisfying the relational precondition. Judgments in pRHL have the form $\vdash t_1 \sim t_2 : R_0 \Rightarrow R_1$ where R_0, R_1 are two relations on states (the pre- and postcondition, respectively) and t_1, t_2 are the two programs to be compared. Such a judgment can be encoded in Bluebell as:

$$\lfloor R_0 \rfloor \vdash \exists \mu.\, \boldsymbol{C}_\mu\, \boldsymbol{s}.\, (St(\boldsymbol{s}) \land \mathbf{wp}\,[1\colon t_1, 2\colon t_2]\,\{\lfloor R_1 \rfloor\}) \quad \text{where} \quad St(\boldsymbol{s}) \triangleq \lceil \mathsf{x}\langle i\rangle = \boldsymbol{s}(\mathsf{x}\langle i\rangle) \rceil_{\mathsf{x}\langle i\rangle \in I \times \mathbb{X}} \tag{7}$$

As the input state is always conditioned, and the precondition is always a relational lifting, one is always in the position of applying C-CONS to eliminate the implicit conditioning of the lifting and the one wrapping the WP, reducing the problem to a goal where the input state is deterministic (and thus where the primitive rules of WP laws apply without need for further conditioning). As noted in Section 2.4, LHC-style WPs allow us to lift our unary WP rules to binary with little effort.

An interesting property of the encoding in (7) is that anything of the form $\boldsymbol{C}_\mu\, \boldsymbol{s}.\, (St(\boldsymbol{s}) \land \dots)$ has ownership of the full store (as it conditions on every variable). We observe that WPs (of any arity) which have this property enjoy an extremely powerful rule. Let $\mathsf{own}_{\mathbb{X}} \triangleq \forall \mathsf{x}\langle i\rangle \in I \times \mathbb{X}.\, \mathsf{own}(\mathsf{x}\langle i\rangle)$. The following is a valid (primitive) rule in Bluebell:

C-WP-SWAP

$$\boldsymbol{C}_\mu\, v.\, \mathbf{wp}\, \boldsymbol{t}\, \{Q(v)\} \land \mathsf{own}_{\mathbb{X}} \vdash \mathbf{wp}\, \boldsymbol{t}\, \left\{\boldsymbol{C}_\mu\, v.\, Q(v)\right\}$$

Rule C-WP-SWAP, allows the shift of the conditioning on the input to the conditioning of the output. This rule provides a powerful way to make progress in lifting a conditional statement to an unconditional one. To showcase C-WP-SWAP, consider the two programs in Fig. 7, which are equivalent: if we couple the x in both programs, the other two samplings can be coupled under conditioning on x. Formally, let $P \Vdash Q \triangleq P \land \mathsf{own}_{\mathbb{X}} \vdash Q \land \mathsf{own}_{\mathbb{X}}$. We process the two assignments to x, which we can couple $\mathsf{x}\langle 1\rangle \sim d_0 * \mathsf{x}\langle 2\rangle \sim d_0 \vdash \boldsymbol{C}_{d_0}\, v.\, (\lceil \mathsf{x}\langle 1\rangle = v \rceil \land \lceil \mathsf{x}\langle 2\rangle = v \rceil)$. Then, let t_1 (t_2) be the rest of `prog1` (`prog2`). We can then derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\forall v.\, \lceil \mathsf{x}\langle 1\rangle = v \rceil \land \lceil \mathsf{x}\langle 2\rangle = v \rceil \Vdash \mathbf{wp}\,[1\colon t_1, 2\colon t_2] \left\{\begin{matrix} \lfloor \mathsf{x}\langle 1\rangle = \mathsf{x}\langle 2\rangle \rfloor * \mathsf{y}\langle 1\rangle \sim d_1(v) * \mathsf{y}\langle 2\rangle \sim d_1(v) * \\ \mathsf{z}\langle 1\rangle \sim d_2(v) * \mathsf{z}\langle 2\rangle \sim d_2(v) \end{matrix}\right\}}{\forall v.\, \lceil \mathsf{x}\langle 1\rangle = v \rceil \land \lceil \mathsf{x}\langle 2\rangle = v \rceil \Vdash \mathbf{wp}\,[1\colon t_1, 2\colon t_2]\,\{\lfloor \mathsf{x}\langle 1\rangle = \mathsf{x}\langle 2\rangle \rfloor * \lfloor \mathsf{y}\langle 1\rangle = \mathsf{y}\langle 2\rangle \rfloor * \lfloor \mathsf{z}\langle 1\rangle = \mathsf{z}\langle 2\rangle \rfloor\}}\ \textsc{coupling}}{\forall v.\, \lceil \mathsf{x}\langle 1\rangle = v \rceil \land \lceil \mathsf{x}\langle 2\rangle = v \rceil \Vdash \mathbf{wp}\,[1\colon t_1, 2\colon t_2]\,\{\lfloor \mathsf{x}\langle 1\rangle = \mathsf{x}\langle 2\rangle \land \mathsf{y}\langle 1\rangle = \mathsf{y}\langle 2\rangle \land \mathsf{z}\langle 1\rangle = \mathsf{z}\langle 2\rangle \rfloor\}}\ \textsc{rl-merge}}{\boldsymbol{C}_{d_0}\, v.\, (\lceil \mathsf{x}\langle 1\rangle = v \rceil \land \lceil \mathsf{x}\langle 2\rangle = v \rceil) \Vdash \boldsymbol{C}_{d_0}\, v.\, \mathbf{wp}\,[1\colon t_1, 2\colon t_2]\,\{\lfloor \mathsf{x}\langle 1\rangle = \mathsf{x}\langle 2\rangle \land \mathsf{y}\langle 1\rangle = \mathsf{y}\langle 2\rangle \land \mathsf{z}\langle 1\rangle = \mathsf{z}\langle 2\rangle \rfloor\}}\ \textsc{c-cons}}{\boldsymbol{C}_{d_0}\, v.\, (\lceil \mathsf{x}\langle 1\rangle = v \rceil \land \lceil \mathsf{x}\langle 2\rangle = v \rceil) \Vdash \mathbf{wp}\,[1\colon t_1, 2\colon t_2]\,\{\boldsymbol{C}_{d_0}\, v.\, \lfloor \mathsf{x}\langle 1\rangle = \mathsf{x}\langle 2\rangle \land \mathsf{y}\langle 1\rangle = \mathsf{y}\langle 2\rangle \land \mathsf{z}\langle 1\rangle = \mathsf{z}\langle 2\rangle \rfloor\}}\ \textsc{c-wp-swap}}{\boldsymbol{C}_{d_0}\, v.\, (\lceil \mathsf{x}\langle 1\rangle = v \rceil \land \lceil \mathsf{x}\langle 2\rangle = v \rceil) \Vdash \mathbf{wp}\,[1\colon t_1, 2\colon t_2]\,\{\lfloor \mathsf{x}\langle 1\rangle = \mathsf{x}\langle 2\rangle \land \mathsf{y}\langle 1\rangle = \mathsf{y}\langle 2\rangle \land \mathsf{z}\langle 1\rangle = \mathsf{z}\langle 2\rangle \rfloor\}}\ \textsc{rl-convex}$$

Where the top triple can be easily derived using standard steps. Reading it from bottom to top, we start by invoking convexity of relational lifting to introduce a conditioning modality in the postcondition matching the one in the precondition. Rule C-WP-SWAP allows us to bring the whole WP under the modality, allowing rule c-cons to remove it on both sides. From then it is a matter of establishing and combining the couplings on $\mathtt{y}$ and $\mathtt{z}$. Note that these couplings are only possible because the coupling on $\mathtt{x}$ made the parameters of d_1 and of d_2 coincide on both indices. In Section 5.5 we show this kind of derivation can be useful for unary reasoning too.

While the $\mathsf{own}_{\mathbb{X}}$ condition is restricting, without it the rule is unsound in the current model. We leave it as future work to study whether there is a model that validates this rule without requiring $\mathsf{own}_{\mathbb{X}}$.

5.2 One Time Pad Revisited

In Section 2, we prove the encrypt program correct relationally (missing details are in [Bao et al. 2024]). An alternative way of stating and proving the correctness of encrypt is to establish that in the output distribution c and m are independent, which can be expressed as the *unary* goal (also studied in [Barthe et al. 2019]): $(\boldsymbol{p}) \vdash \mathbf{wp}\,[1{:}\,\mathtt{encrypt()}]\,\{\mathtt{c}\langle 1\rangle \sim \mathsf{Ber}_{1/2} * \mathtt{m}\langle 1\rangle \sim \mathsf{Ber}_p\}$ (where $\boldsymbol{p} = [\mathtt{k}\langle 1\rangle{:}\,1, \mathtt{m}\langle 1\rangle{:}\,1, \mathtt{c}\langle 1\rangle{:}\,1]$). The triple states that after running encrypt, the ciphertext c is distributed as a fair coin, and—importantly—is *not* correlated with the plaintext in m. The PSL proof in [Barthe et al. 2019] performs some of the steps within the logic, but needs to carry out some crucial entailments at the meta-level, which is a symptom of unsatisfactory abstractions (**RQ1**). The same applies to the Lilac proof in [Li et al. 2023b] which requires ad-hoc lemmas proven on the semantic model. The stumbling block is proving the valid entailment:

$$\mathtt{k}\langle 1\rangle \sim \mathsf{Ber}_{½} * \mathtt{m}\langle 1\rangle \sim \mathsf{Ber}_p * \lceil \mathtt{c}\langle 1\rangle = \mathtt{k}\langle 1\rangle \mathbin{\mathtt{xor}} \mathtt{m}\langle 1\rangle \rceil \vdash \mathtt{m}\langle 1\rangle \sim \mathsf{Ber}_p * \mathtt{c}\langle 1\rangle \sim \mathsf{Ber}_{½}$$

In Bluebell we can prove the entailment in two steps: (1) we condition on m and k to compute the result of the xor operation and obtain that c is distributed as $\mathsf{Ber}_{½}$; (2) we carefully eliminate the conditioning while preserving the independence of m and c.

The first step starts by conditioning on m and k and proceeds as follows:

$$\begin{aligned}
&\boldsymbol{C}_{\mathsf{Ber}_p}\, m.\, \big(\lceil \mathtt{m}\langle 1\rangle = m \rceil * \boldsymbol{C}_{\mathsf{Ber}_{½}}\, k.\, (\lceil \mathtt{k}\langle 1\rangle = k \rceil * \lceil \mathtt{c}\langle 1\rangle = k \mathbin{\mathtt{xor}} m \rceil)\big) \\
\vdash\ &\boldsymbol{C}_{\mathsf{Ber}_p}\, m.\, \left(\lceil \mathtt{m}\langle 1\rangle = m \rceil * \begin{cases} \boldsymbol{C}_{\mathsf{Ber}_{½}}\, k.\, \lceil \mathtt{c}\langle 1\rangle = k \rceil & \text{if } m = 0 \\ \boldsymbol{C}_{\mathsf{Ber}_{½}}\, k.\, \lceil \mathtt{c}\langle 1\rangle = \neg k \rceil & \text{if } m = 1 \end{cases}\right) && (\text{c-cons}) \\
\vdash\ &\boldsymbol{C}_{\mathsf{Ber}_p}\, m.\, \big(\lceil \mathtt{m}\langle 1\rangle = m \rceil * \boldsymbol{C}_{\mathsf{Ber}_{½}}\, k.\, \lceil \mathtt{c}\langle 1\rangle = k \rceil\big) && (\text{c-transf})
\end{aligned}$$

The crucial entailment is the application of c-transf to the $m = 1$ branch, by using negation as the bijection (which satisfies the premises of the rules since $\mathsf{Ber}_{½}$ is unbiased).

The second step uses the following primitive rule of Bluebell:

$$\begin{array}{l}\text{prod-split}\\ (E_1\langle i\rangle, E_2\langle i\rangle) \sim \mu_1 \otimes \mu_2 \vdash E_1\langle i\rangle \sim \mu_1 * E_2\langle i\rangle \sim \mu_2\end{array}$$

with which we can prove:

$$\begin{aligned}
&\boldsymbol{C}_{\mathsf{Ber}_p}\, m.\, \big(\lceil \mathtt{m}\langle 1\rangle = m \rceil * \boldsymbol{C}_{\mathsf{Ber}_{½}}\, k.\, \lceil \mathtt{c}\langle 1\rangle = k \rceil\big) \\
\vdash\ &\boldsymbol{C}_{\mathsf{Ber}_p}\, m.\, \boldsymbol{C}_{\mathsf{Ber}_{½}}\, k.\, \lceil \mathtt{m}\langle 1\rangle = m \land \mathtt{c}\langle 1\rangle = k \rceil && (\text{c-frame}) \\
\vdash\ &\boldsymbol{C}_{\mathsf{Ber}_p \otimes \mathsf{Ber}_{½}}(m, k).\, \lceil (\mathtt{m}\langle 1\rangle, \mathtt{c}\langle 1\rangle) = (m, k) \rceil && (\text{c-fuse}) \\
\vdash\ &(\mathtt{m}\langle 1\rangle, \mathtt{c}\langle 1\rangle) \sim (\mathsf{Ber}_p \otimes \mathsf{Ber}_{½}) && (\text{c-unit-r}) \\
\vdash\ &\mathtt{m}\langle 1\rangle \sim \mathsf{Ber}_p * \mathtt{c}\langle 1\rangle \sim \mathsf{Ber}_{½} && (\text{prod-split})
\end{aligned}$$

As this is a common manipulation needed to extract unconditional independence from a conditional fact, we can formulate it as the more general derived rule

C-EXTRACT

$$\boldsymbol{C}_{\mu_1} v_1.\, \big(\lceil E_1\langle i\rangle = v_1 \rceil * E_2\langle i\rangle \sim \mu_2\big) \vdash E_1\langle i\rangle \sim \mu_1 * E_2\langle i\rangle \sim \mu_2$$

5.3 Markov Blankets

In probabilistic reasoning, introducing conditioning is easy, but deducing unconditional facts from conditional ones is not immediate. The same applies to the joint conditioning modality: by design, one cannot eliminate it for free. Crucial to Bluebell's expressiveness is the inclusion of rules that can soundly derive unconditional information from conditional assertions.

We use the concept of a *Markov Blanket*—a very common tool in Bayesian reasoning for simplifying conditioning—to illustrate Bluebell's expressiveness (**RQ1** and **RQ2**). Intuitively, Markov blankets identify a set of variables that affect the distribution of a random variable *directly*: this is useful because by conditioning on those variables we can remove conditional connection between the random variable and all the variables on which it *indirectly* depends.

For concreteness, consider the program $\mathsf{x1} \mathrel{:\approx} d_1;\ \ \mathsf{x2} \mathrel{:\approx} d_2(\mathsf{x1});\ \ \mathsf{x3} \mathrel{:\approx} d_3(\mathsf{x2})$. The program describes a Markov chain of three variables. One way of interpreting this pattern is that the joint output distribution is described by the program as a product of conditional distributions: the distribution over $\mathsf{x2}$ is described conditionally on $\mathsf{x1}$, and the one of $\mathsf{x3}$ conditionally on $\mathsf{x2}$. This kind of dependencies are ubiquitous in, for instance, hidden Markov models and Bayesian network representations of distributions.

A crucial tool for the analysis of such models is the concept of a Markov Blanket of a variable x: the set of variables that are direct dependencies of x. Clearly $\mathsf{x3}$ depends on $\mathsf{x2}$ and, indirectly, on $\mathsf{x1}$. However, Markov chains enjoy the memorylessness property: when fixing a variable in the chain, the variables that follow it are independent from the variables that preceded it. For our example this means that if we condition on $\mathsf{x2}$, then $\mathsf{x1}$ and $\mathsf{x3}$ are independent (i.e. we can ignore the indirect dependencies).

In Bluebell we can characterize the output distribution with the assertion

$$\boldsymbol{C}_{d_1} v_1.\, \Big(\lceil \mathsf{x1} = v_1 \rceil * \boldsymbol{C}_{d_2(v_1)} v_2.\, \big(\lceil \mathsf{x2} = v_2 \rceil * \mathsf{x3} \sim d_3(v_2)\big)\Big)$$

Note how this postcondition represents the output distribution as implicitly as the program does. We want to transform the assertion into:

$$\boldsymbol{C}_{\mu_2} v_2.\, \big(\lceil \mathsf{x2} = v_2 \rceil * \mathsf{x1} \sim \mu_1(v_2) * \mathsf{x3} \sim d_3(v_2)\big)$$

for appropriate μ_2 and μ_1. This isolates the conditioning to the direct dependency of $\mathsf{x1}$ and keeps full information about $\mathsf{x3}$, available for further manipulation down the line.

In probability theory, the proof of memorylessness is an application of Bayes' law: we are computing the distribution of $\mathsf{x1}$ conditioned on $\mathsf{x2}$, from the distribution of $\mathsf{x2}$ conditioned on $\mathsf{x1}$.

In Bluebell we can produce the transformation using the joint conditioning rules, in particular the right-to-left direction of C-FUSE and the primitive rule that is behind its left-to-right direction:

C-UNASSOC

$$\boldsymbol{C}_{\mathbf{bind}(\mu,\kappa)}\, w.\, K(w) \vdash \boldsymbol{C}_{\mu}\, v.\, \boldsymbol{C}_{\kappa(v)}\, w.\, K(w)$$

Using these we can prove:

$$\begin{aligned}
& \boldsymbol{C}_{d_1}\, v_1.\, \Big(\lceil \mathtt{x1} = v_1 \rceil * \boldsymbol{C}_{d_2(v_1)}\, v_2.\, \big(\lceil \mathtt{x1} = v_2 \rceil * \mathtt{x3} \sim d_3(v_2)\big)\Big) && \\
\vdash\ & \boldsymbol{C}_{d_1}\, v_1.\, \Big(\boldsymbol{C}_{d_2(v_1)}\, v_2.\, \big(\lceil \mathtt{x1} = v_1 \rceil * \lceil \mathtt{x1} = v_2 \rceil * \mathtt{x3} \sim d_3(v_2)\big)\Big) && \text{(C-FRAME)} \\
\vdash\ & \boldsymbol{C}_{\mu_0}(v_1, v_2).\, \big(\lceil \mathtt{x1} = v_1 \rceil * \lceil \mathtt{x2} = v_2 \rceil * \mathtt{x3} \sim d_3(v_2)\big) && \text{(C-FUSE)} \\
\vdash\ & \boldsymbol{C}_{\mu_2}\, v_2.\, \Big(\boldsymbol{C}_{\mu_1(v_2)}\, v_1.\, \big(\lceil \mathtt{x1} = v_1 \rceil * \lceil \mathtt{x2} = v_2 \rceil * \mathtt{x3} \sim d_3(v_2)\big)\Big) && \text{(C-UNASSOC)} \\
\vdash\ & \boldsymbol{C}_{\mu_2}\, v_2.\, \Big(\lceil \mathtt{x2} = v_2 \rceil * \boldsymbol{C}_{\mu_1(v_2)}\, v_1.\, \big(\lceil \mathtt{x1} = v_1 \rceil * \mathtt{x3} \sim d_3(v_2)\big)\Big) && \text{(SURE-STR-CONVEX)} \\
\vdash\ & \boldsymbol{C}_{\mu_2}\, v_2.\, \big(\lceil \mathtt{x2} = v_2 \rceil * \mathtt{x1} \sim \mu_1(v_2) * \mathtt{x3} \sim d_3(v_2)\big) && \text{(C-EXTRACT)}
\end{aligned}$$

where $d_1 \prec d_2 = \mu_0 = \mathbf{bind}(\mu_2, \mu_1)$. The existence of such μ_2 and μ_1 is a simple application of Bayes' law: $\mu_2(v_2) = \sum_{v_1 \in \mathbb{V}} \mu_0(v_1, v_2)$, and $\mu_1(v_2)(v_1) = \frac{\mu_0(v_1, v_2)}{\mu_2(v_2)}$. We see the ability of Bluebell to perform these manipulations as evidence that joint conditioning and independence form a sturdy abstraction over the semantic model (**RQ1**). The amount of meta-reasoning required to manipulate the distributions indexing the conditioning modality are minimal and localized, and offer a good entry-point to inject facts about distributions without interfering with the rest of the proof context.

5.4 Multi-party Secure Computation

In *multi-party secure computation*, the goal is to for N parties to compute a function $f(x_1, \ldots, x_N)$ of some private data x_i owned by each party i, without revealing any more information about x_i than the output of f would reveal if computed centrally by a trusted party. For example, if f is addition, a secure computation of f can be used to compute the total number of votes without revealing who voted positively: some information would leak (e.g. if the total is non-zero then *somebody* voted positively) but only what is revealed by knowing the total and nothing more.

To achieve this objective, multi-party secure addition (MPSAdd) works by having the parties break their secret into N *secret shares* which individually look random, but the sum of which amounts to the original secret. These secret shares are then distributed to the other parties so that each party knows an incomplete set of shares of the other parties. Yet, each party can reliably compute the result of the function by computing a function of the received shares.

As it is very often the case, there is no single "canonical" way of specifying this kind of security property. For MPSAdd, for instance, we can formalize security (focusing on the perspective of party 1) in two ways: as a unary or as a relational specification.

The *unary specification* says that, conditionally on the secret of party 1 and the sum of the other secrets, all the values received by 1 (we call this the *view* of 1) are independent from the secrets of the other parties. Roughly:

$$(\mathtt{x}_1, \mathtt{x}_2, \mathtt{x}_3)\langle 1 \rangle \sim \mu_0 \vdash \mathbf{wp}\, [1{:}\, \mathtt{MPSAdd}] \left\{ \exists \mu.\, \boldsymbol{C}_\mu\, (v, s).\, \begin{pmatrix} \lceil \mathtt{x}_1\langle 1 \rangle = v \land (\mathtt{x}_2 + \mathtt{x}_3)\langle 1 \rangle = s \rceil * \\ \mathtt{own}(\mathtt{view}_1\langle 1 \rangle) * \mathtt{own}(\mathtt{x}_2\langle 1 \rangle, \mathtt{x}_3\langle 1 \rangle) \end{pmatrix} \right\}$$

where μ_0 is an arbitrary distribution of the three secrets. Notice how conditioning nicely expresses that the acceptable leakage is just the sum.

The *relational specification* says that when running the program from two initial states differing only in the secrets of the other parties, but not in their sum, the views of party i would be distributed in the same way. Roughly:

$$\left\lfloor \begin{matrix} \mathtt{x}_1\langle 1 \rangle = \mathtt{x}_1\langle 2 \rangle \\ (\mathtt{x}_2 + \mathtt{x}_3)\langle 1 \rangle = (\mathtt{x}_2 + \mathtt{x}_3)\langle 2 \rangle \end{matrix} \right\rfloor \vdash \mathbf{wp} \begin{bmatrix} 1{:}\ \mathtt{MPSAdd} \\ 2{:}\ \mathtt{MPSAdd} \end{bmatrix} \left\{ \lfloor \mathtt{view}_1\langle 1 \rangle = \mathtt{view}_1\langle 2 \rangle \rfloor \right\}$$

The two specifications look quite different and also suggest quite different proof strategies: the unary judgment suggests a proof by manipulating independence and conditioning; the relational one hints at a proof by relational lifting. Depending on the program, each of these strategies could have their merits. As a first contribution, we show that Bluebell can not only specify in both styles (**RQ1**), but also provide *proofs* in both styles (**RQ4**).

Having two very different specifications for the same security goal, however begs the question: are they equivalent? After all, as the *prover* of the property one might prefer one proof style over the other, but as a *consumer* of the specification the choice might be dictated by the proof context that needs to use the specification for proving a global goal. To decouple the proof strategy from the uses of the specification, we would need to be able to convert one specification into the other *within* the logic, thus sparing the prover from having to forsee which specification a proof context might need in the future.

Our second key result is that in fact the equivalence between the unary and the relational specification can be proven in Bluebell. This is enabled by the powerful joint conditioning rules and the encoding of relational lifting as joint conditioning. This is remarkable as this type of result has always been justified entirely at the level of the semantic model in other logics (e.g. pRHL, Lilac). This illustrates the fitness of Bluebell as a tool for abstract meta-level reasoning (**RQ1**).

In [Bao et al. 2024] we provide Bluebell proofs for: (1) the unary specification; (2) the relational specification (independently of the unary proof); (3) the equivalence of the two specifications. Although the third item would spare us from proving one of the first two, we provide direct proofs in the two styles to provide a point of comparison between them.

5.5 Von Neumann Extractor

A randomness extractor is a mechanism that transforms a stream of "low-quality" randomness sources into a stream of "high-quality" randomness sources. The von Neumann extractor [von Neumann 1951] is perhaps the earliest instance of such mechanism, and it converts a stream of independent coins with the same bias p into a stream of independent *fair* coins. Verifying the correctness of the extractor requires careful reasoning under conditioning, and showcases the use of rule c-wp-swap in a unary setting (**RQ2** and **RQ4**).

We can model the extractor, up to $N \in \mathbb{N}$ iterations, in our language[3] as shown in Fig. 5. The program repeatedly flips two biased coins, and outputs the outcome of the first coin if the outcomes where different, otherwise it retries. As an example, we prove in Bluebell that the bits produced in `out` are independent fair coin flips. Formally, for ℓ produced bits, we want the following to hold:

$$Out_\ell \triangleq \texttt{out}[0]\langle 1\rangle \sim \mathsf{Ber}_{1/2} * \cdots * \texttt{out}[\ell-1]\langle 1\rangle \sim \mathsf{Ber}_{1/2}.$$

To know how many bits were produced, however, we need to condition on `len` obtaining the specification (recall $P \Vdash Q \triangleq P \land \mathsf{own}_{\mathbb{X}} \vdash Q \land \mathsf{own}_{\mathbb{X}}$):

$$\Vdash \mathbf{wp}\,[1{:}\,\mathsf{vn}(N)]\,\big\{\exists\mu.\,\boldsymbol{C}_\mu\,\ell.\,\big(\lceil \texttt{len}\langle 1\rangle = \ell \le N \rceil * Out_\ell\big)\big\}$$

The postcondition straightforwardly generalizes to a loop invariant

$$P(i) = \exists\mu.\,\boldsymbol{C}_\mu\,\ell.\,\big(\lceil \texttt{len}\langle 1\rangle = \ell \le i \rceil * Out_\ell\big)$$

The main challenge in the example is handling the if-then statement. Intuitively we want to argue that if $\texttt{coin}_1 \neq \texttt{coin}_2$, the two coins would have either values $(0,1)$ or $(1,0)$, and both of these outcomes have probability $p(1-p)$; therefore, conditionally on the 'if' guard being true, $\texttt{coin}_1$ is a fair coin.

[3] While technically our language does not support arrays, they can be easily encoded as a collection of N variables.

Two features of Bluebell are crucial to implement the above intuition. The first is the ability of manipulating conditioning given by the joint conditioning rules. At the entry point of the if-then statement in Fig. 5 we obtain $P(i) * \texttt{coin}_1\langle 1\rangle \sim \mathrm{Ber}_p * \texttt{coin}_2\langle 1\rangle \sim \mathrm{Ber}_p$. Using Bluebell's rules we can easily derive $P(i) * (\texttt{coin}_1 \neq \texttt{coin}_2, \texttt{coin}_1)\langle 1\rangle \sim \mu_0$ for some μ_0. The main insight of the algorithm then can be expressed as the fact that $\mu_0 = \beta \prec \kappa$ for some $\beta : \mathbb{D}(\{0,1\})$ which is the distribution of $\texttt{coin}_1 \neq \texttt{coin}_2$, and some κ describing the distribution of the first coin in the two cases, which we know is such that $\kappa(1) = \mathrm{Ber}_{1/2}$. Then, thanks to c-unit-r and c-fuse, we obtain:

```
def vn($N$):
  len := 0
  repeat $N$:
    $\texttt{coin}_1$ :≈ Ber($p$)
    $\texttt{coin}_2$ :≈ Ber($p$)
    if $\texttt{coin}_1 \neq \texttt{coin}_2$ then:
      out[len] := $\texttt{coin}_1$
      len := len+1
```

Fig. 5. Von Neumann extractor.

$$\begin{aligned}&(\texttt{coin}_1 \neq \texttt{coin}_2, \texttt{coin}_1)\langle 1\rangle \sim (\beta \prec \kappa)\\ \vdash\ &\boldsymbol{C}_\beta\, b.\, \big(\lceil(\texttt{coin}_1 \neq \texttt{coin}_2)\langle 1\rangle = b\rceil * \ulcorner b = 1 \urcorner \Rightarrow \texttt{coin}_1\langle 1\rangle \sim \mathrm{Ber}_{1/2}\big)\end{aligned}$$

Then, if we could reason about the 'then' branch under conditioning, since the guard $\texttt{coin}_1 \neq \texttt{coin}_2$ implies $b = 1$ we would obtain $\texttt{coin}_1\langle 1\rangle \sim \mathrm{Ber}_{1/2}$, which is the key to the proof. The ability of reasoning under conditioning is the second feature of Bluebell which unlocks the proof. In this case, the step is driven by rule c-wp-swap, which allows us to prove the if-then statement by case analysis on b.

5.6 Monte Carlo Algorithms

By elaborating on the Monte Carlo example of Section 1, we want to show the fitness of Bluebell as a program logic (**RQ4**) and its specific approach for dealing with the structure of a program. Recall the example in Figure 1 and the goal outlined in Section 1 of comparing the accuracy of the two Monte Carlo algorithms BETW_SEQ and BETW. This goal can be encoded as

$$\left(\begin{matrix}\lceil \texttt{l}\langle 1\rangle = \texttt{r}\langle 1\rangle = 0\rceil\ * \\ \lceil \texttt{l}\langle 2\rangle = \texttt{r}\langle 2\rangle = 0\rceil\end{matrix}\right)@\boldsymbol{p} \vdash \mathbf{wp}\left[\begin{matrix}1\!: \texttt{BETW\_SEQ}(x,S)\\ 2\!: \texttt{BETW}(x,S)\end{matrix}\right]\left\{\lfloor \texttt{d}\langle 1\rangle \leq \texttt{d}\langle 2\rangle \rfloor\right\}$$

(where $\boldsymbol{p}$ contains full permissions for all the variables) which, through the relational lifting, states that it is more likely to get a positive answer from BETW than from BETW_SEQ. The challenge is implementing the intuitive relational argument sketched in Section 1, in the presence of very different looping structures. More precisely, we want to compare the sequential composition of two loops $l_1 = (\textbf{repeat}\ N\ t_{\mathsf{A}}; \textbf{repeat}\ N\ t_{\mathsf{B}})$ with a single loop $l_2 = \textbf{repeat}\ (2N)\ t$ considering the N iterations of t_{A} in lockstep with the first N iterations of l_2, and the N iterations of t_{B} with the remaining N iterations of l_2. It is not possible to perform such proof purely in pRHL, which can only handle loops that are perfectly aligned, and tools based on pRHL overcome this limitation by offering a number of code transformations, proved correct externally to the logic, with which one can rewrite the loops so that they syntactically align. In this case such a transformation could look like $\textbf{repeat}\ (M+N)\ t \equiv \textbf{repeat}\ M\ t; \textbf{repeat}\ N\ t$, using which one can rewrite l_2 so it aligns with the two shorter loops. What Bluebell can achieve is to avoid the use of such ad-hoc syntactic transformations, and produce a proof structured in two steps: first, one can prove, *within the logic*, that it is sound to align the loops as described; and then proceed with the proof of the aligned loops.

```
def BETW_MIX(x,S):
  repeat N:
    p :≈ μS;  l := l || p ≤ x
    q :≈ μS;  r := r || q ≥ x
  d := r && l
```

Fig. 6. A variant of the BETW program.

```
def prog1:        def prog2:
  x :≈ d0           x :≈ d0
  y :≈ d1(x)        z :≈ d2(x)
  z :≈ d2(x)        y :≈ d1(x)
```

Fig. 7. Conditional Swapping

The key idea is that the desired alignment of loops can be expressed as a (derived) rule, encoding the net effect of the syntactic loop splitting, without having to manipulate the syntax:

wp-loop-split

$$\frac{\begin{array}{c} P_1(N_1) \vdash P_2(0) \\ \forall i < N_1.\, P_1(i) \vdash \mathbf{wp}\,[1{:}\, t_1, 2{:}\, t]\,\{P_1(i+1)\} \\ \forall j < N_2.\, P_2(j) \vdash \mathbf{wp}\,[1{:}\, t_2, 2{:}\, t]\,\{P_2(j+1)\} \end{array}}{P_1(0) \vdash \mathbf{wp}\,[1{:}\, (\texttt{repeat}\ N_1\ t_1\,\texttt{;}\,\texttt{repeat}\ N_2\ t_2), 2{:}\, \texttt{repeat}\ (N_1+N_2)\ t]\,\{P_2(N_2)\}}$$

The rule considers two programs: a sequence of two loops, and a single loop with the same cumulative number of iterations. It asks the user to produce two relational loop invariants P_1 and P_2 which are used to relate N_1 iterations of t_1 and t together, and N_2 iterations of t_2 and t together.

Crucially, such rule is *derivable* from the primitive rules of looping of Bluebell:

wp-loop

$$\frac{\forall i < n.\, P(i) \vdash \mathbf{wp}\,[j{:}\, t]\,\{P(i+1)\}}{P(0) \vdash \mathbf{wp}\,[j{:}\, \texttt{repeat}\ n\ t]\,\{P(n)\}}\ n \in \mathbb{N}$$

wp-loop-unf

$$\begin{array}{l} \mathbf{wp}\,[i{:}\, \texttt{repeat}\ n\ t]\,\{\mathbf{wp}\,[i{:}\, t]\,\{Q\}\} \\ \vdash \mathbf{wp}\,[i{:}\, \texttt{repeat}\ (n+1)\ t]\,\{Q\} \end{array}$$

Rule wp-loop is a standard unary invariant-based rule; wp-loop-unf simply reflects the semantics of a loop in terms of its unfoldings. Using these we can prove wp-loop-split avoiding semantic reasoning all together, and fully generically on the loop bodies, allowing it to be reused in any situation fitting the pattern.

In our example, we can prove our goal by instanting it with the loop invariants:

$$P_1(i) \triangleq \lfloor \mathtt{r}\langle 1\rangle \le \mathtt{r}\langle 2\rangle \wedge \mathtt{l}\langle 1\rangle = 0 \le \mathtt{l}\langle 2\rangle \rfloor \qquad P_2(j) \triangleq \lfloor \mathtt{r}\langle 1\rangle \le \mathtt{r}\langle 2\rangle \wedge \mathtt{l}\langle 1\rangle \le \mathtt{l}\langle 2\rangle \rfloor$$

This handling of structural differences as derived proof patterns is more powerful than syntactic transformations: it can, for example, handle transformations that are sound only under some assumptions about state. To show an instance of this, we consider a variant of the previous example: BETW_MIX (in Fig. 6) is another variant of BETW_SEQ which still makes $2N$ samples but interleaves sampling for the minimum and for the maximum. We want to prove that this is equivalent to BETW_SEQ. Letting $\boldsymbol{p}$ contain full permissions for the relevant variables, the goal is

$$P_0@\boldsymbol{p} \vdash \mathbf{wp}\,[1{:}\, \mathtt{BETW\_SEQ}(x,S), 2{:}\, \mathtt{BETW\_MIX}(x,S)]\,\{\lfloor \mathtt{d}\langle 1\rangle = \mathtt{d}\langle 2\rangle \rfloor\}$$

with $P_0 = \lceil \mathtt{l}\langle 1\rangle = \mathtt{r}\langle 1\rangle = 0 \rceil * \lceil \mathtt{l}\langle 2\rangle = \mathtt{r}\langle 2\rangle = 0 \rceil$.

Call t^1_{M} and t^2_{M} the first and second half of the body of the loop of BETW_MIX, respectively. The strategy is to consider together one execution of t_{A} (the body of the loop of AboveMin), and t^1_{M}; and one of t_{B} (of BelowMax), and t^2_{M}. The strategy relies on the observation that every iteration of the three loops is *independent* from the others. To formalize the proof idea we thus first prove a derived proof pattern encoding the desired alignment, which we can state for generic t_1, t_2, t'_1, t'_2:

wp-loop-mix

$$\frac{\forall i < N.\, P_1(i) \vdash \mathbf{wp}\,[1{:}\,t_1, 2{:}\,t'_1]\,\{P_1(i+1)\} \qquad \forall i < N.\, P_2(i) \vdash \mathbf{wp}\,[1{:}\,t_2, 2{:}\,t'_2]\,\{P_2(i+1)\}}{P_1(0) * P_2(0) \vdash \mathbf{wp}\,[1{:}\,(\mathtt{repeat}\ N\ t_1\,;\mathtt{repeat}\ N\ t_2), 2{:}\,\mathtt{repeat}\ N\ (t'_1; t'_2)]\,\{P_1(N) * P_2(N)\}}$$

The rule matches on two programs: a sequence of two loops, and a single loop with a body split into two parts. The premises require a proof that t_1 together with t'_1 (the first half of the body of the second loop) preserve the invariant P_1; and that the same is true for t_2 and t'_2 with respect to an invariant P_2. The precondition $P_1(0) * P_2(0)$ in the conclusion ensures that the two loop invariants are independent. The rule wp-loop-mix can be again entirely derived from Bluebell's primitive rules. We can then apply it to our example using as invariants $P_1 \triangleq \lfloor \mathtt{l}\langle 1\rangle = \mathtt{l}\langle 2\rangle \rfloor$ and $P_2 \triangleq \lfloor \mathtt{r}\langle 1\rangle = \mathtt{r}\langle 2\rangle \rfloor$. Then, rl-merge closes the proof.

6 RELATED WORK

Research on deductive verification of probabilistic programs has developed a wide range of techniques that employ *unary* and *relational* styles of reasoning. Bluebell advances the state of the art in both styles, by coherently unifying the strengths of both. We limit our comparison here to deductive techniques only, and focus most of our attention on explaining how Bluebell offers new reasoning tools compared to these.

Unary-style Reasoning. Early work in this line focuses more on analyzing marginal distributions and probabilities, and features like harnessing the power of probabilistic independence and conditioning have been more recently added to make more expressive program logics [Bao et al. 2022; Barthe et al. 2018, 2019; Li et al. 2023a; Ramshaw 1979; Rand and Zdancewic 2015].

Much work in this line has been inspired by *Separation Logic* (SL), a powerful tool for reasoning about pointer-manipulating programs, known for its support of *local reasoning* of separated program components [Reynolds 2000]. PSL [Barthe et al. 2019] was the first logic to present a SL model for reasoning about the probabilistic independence of program variables, which facilitates modular reasoning about independent components within a probabilistic program. In [Bao et al. 2021] and [Bao et al. 2022] SL variants are used for reasoning about conditional independence and negative dependence, respectively; both are used in algorithm analysis as relaxations of independence.

Lilac. Lilac [Li et al. 2023a] is the most recent addition to this group and introduces a new foundation of probabilistic separation logic based on measure theory. It enables reasoning about independence and conditional independence uniformly in one logic and supports continuous distributions. Bluebell also uses a measure-theory based model, similar to Lilac, although limited to discrete distributions. While Bluebell uses Lilac's independent product as a model of separating conjunction, it differs from Lilac in three aspects: (1) the treatment of ownership, (2) support for mutable state, and (3) the model of conditioning.

Ownership as almost-measurability is required to support inferences like $\mathsf{own}(\mathtt{x}) * \lceil \mathtt{x} = \mathtt{y} \rceil \vdash \mathsf{own}(\mathtt{y})$, which were implicitly used in the first version of Lilac, but were not valid in its model. Li et al. [2023b] fixes the issue by changing the meaning of $\lceil \mathtt{x} = \mathtt{y} \rceil$, while our fix acts on the meaning of ownership (and we see $\lceil E \rceil$ assertions as an instance of regular ownership).

Lilac works with immutable state [Staton 2020], which simplifies reasoning in certain contexts (e.g., the frame rule and the if rule). Bluebell's model supports mutable state through a creative use of permissions, obtaining a clean frame rule, at the cost of some predictable bookkeeping.

The more significant difference with Lilac is however in the definition of the conditioning modality. Lilac's modality $\mathbf{C}_{v \leftarrow E}\, P(v)$ is indexed by a random variable E, and roughly corresponds to the Bluebell assertion $\exists \mu.\, \boldsymbol{C}_{\mu}\, v.\, (\lceil E = v \rceil * P(v))$. The difference is not merely syntactic, and requires changing the model of the modality. For example, Lilac's modality satisfies $\mathbf{C}_{v \leftarrow E}\, P_1(v) \land \mathbf{C}_{v \leftarrow E}\, P_2(v) \vdash \mathbf{C}_{v \leftarrow E}\, (P_1(v) \land P_2(v))$, but the analogous rule $\boldsymbol{C}_{\mu}\, v.\, K_1(v) \land \boldsymbol{C}_{\mu}\, v.\, K_2(v) \vdash \boldsymbol{C}_{\mu}\, v.\, (K_1(v) \land K_2(v))$ (corresponding to c-and without the side condition) is unsound in Bluebell: The meaning of the modalities in the premise ensures the *existence* of two kernels κ_1 and κ_2 supporting K_1 and K_2 respectively, but the conclusion requires the existence of a *single* kernel supporting both K_1 and K_2. Lilac's rule holds because when one conditions on a random variable, the corresponding kernels are unique. We did not find losing this rule limiting. On the other hand, Lilac's conditioning has two key disadvantages: (i) it does not record the distribution of E, losing this information when conditioning, (ii) it does not generalize to the relational setting. Even considering only the unary setting, having access to the distribution μ in fact unlocks a number of new rules (e.g. c-unit-r and c-fuse) that are key to the increased expressivity of Bluebell. In particular, the rules of Bluebell provide a wider arsenal of tools that can convert a conditional assertion back into an unconditional one. This is especially important when conditioning is used as a reasoning tool, regardless of whether the end goal is a conditional statement.

Relational Reasoning. Barthe et al. [2009] extend relational Hoare logic [Benton 2004] to reason about probabilistic programs in a logic called pRHL (probabilistic Relational Hoare Logic). In pRHL, assertions on pairs of deterministic program states are lifted to assertions on pairs of distributions, and on the surface, the logic simply manipulates the deterministic assertions. A number of variants of pRHL were successfully applied to proving various cryptographic protocols and differential privacy algorithms [Barthe et al. 2015, 2009; Hsu 2017; Wang et al. 2019; Zhang and Kifer 2017]. When a natural relational proof for an argument exists, these logics are simple and elegant to use. However, they fundamentally trade expressiveness for ease of use. A persisting problem with them has been that they rely on a strict structural alignment between the order of samples in the two programs. Recall our discussion in Section 2.4 for an example of this that Bluebell can handle. Gregersen et al. [2024] recently proposed Clutch, a logic to prove contextual refinement in a probabilistic higher-order language, where "out of order" couplings between samplings are achieved by using ghost code that pre-samples some assignments, a technique inspired by *prophecy variables* [Jung et al. 2019]. In Section 2 we showed how Bluebell can resolve the issue without ghost code (in the context of first-order imperative programs) by using framing and probabilistic independence creatively. In contrast to Bluebell, Clutch can only express relational properties; it also uses separation but with its classical interpretation as disjointness of deterministic state.

Polaris [Tassarotti and Harper 2019], is an early instance of a probabilistic relational (concurrent) separation logic. However, separation in Polaris is again classic disjointness of state.

Our n-ary WP is inspired by LHC [D'Osualdo et al. 2022], which shows how arity-changing rules (like wp-nest) can accommodate modular and flexible relational proofs of deterministic programs.

Other Techniques. Expectation-based approaches, which reason about expected quantities of probabilistic programs via a weakest-pre-expectation operator that propagates information about expected values backwards through the program, have been classically used to verify randomized algorithms [Aguirre et al. 2021; Kaminski 2019; Kaminski et al. 2016; Kozen 1983; Moosbrugger et al. 2022; Morgan et al. 1996]. These logics offer ergonomic dedicated principles for expectations, but do not aim at unifying principles for analyzing more general classes of properties or proof techniques, like we attempt here. Ellora [Barthe et al. 2018] proposes an assertion-based logic (without separation nor conditioning) to overcome the limitation of working only with expectations.

7 CONCLUSIONS AND FUTURE WORK

Bluebell's journey started as a quest to integrate unary and relational probabilistic reasoning and ended up uncovering joint conditioning as a key fundational tool. Remarkably, to achieve our goal we had to deviate from Lilac's previous proposal in both the definition of conditioning, to enable the encoding of relational lifting, and of ownership (with almost measurability), to resolve an issue with almost sure assertions (recently corrected [Li et al. 2023b] in a different way). In addition, our model supports mutable state without sacrificing expressiveness. One limitation of our current model is lack of support for continuous distributions. Lilac's model and recent advances in it [Li et al. 2024] could suggest a pathway for a continuous extension of Bluebell, but it is unclear if all our rules would be still valid; for example rule c-fuse's soundness hinges on properties of discrete distributions that we could not extend to the general case in an obvious way. Bluebell's encoding of relational lifting and the novel proof principles it uncovered for it are a demonstration of the potential of joint conditioning as a basis for deriving high-level logics on top of an ergonomic core logic. Obvious candidates for such scheme are approximate couplings [Barthe et al. 2012] (which have been used for e.g. differential privacy), and expectation-based calculi (à la Ellora).

ACKNOWLEDGMENTS

We would like to thank Justin Hsu for connecting the authors and the many discussions. We also thank Derek Dreyer and Deepak Garg for the discussions and support. We are grateful to the POPL'25 reviewers for their constructive feedback. Jialu Bao was supported by the NSF Award No. 2153916. Emanuele D'Osualdo was supported by a European Research Council (ERC) Consolidator Grant for the project "PERSIST" under the European Union's Horizon 2020 research and innovation programme (grant No. 101003349). Azadeh Farzan was supported by the National Science and Engineering Research Council of Canada Discovery Grant.

REFERENCES

Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Pierre-Yves Strub. 2017. A relational logic for higher-order programs. *Proceedings of the ACM on Programming Languages* 1, ACM SIGPLAN International Conference on Functional Programming (ICFP), Oxford, England (2017), 1–29.

Alejandro Aguirre, Gilles Barthe, Justin Hsu, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. 2021. A pre-expectation calculus for probabilistic sensitivity. *Proceedings of the ACM on Programming Languages* 5, ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Lisbon, Portugal (2021), 1–28.

Jialu Bao, Simon Docherty, Justin Hsu, and Alexandra Silva. 2021. A bunched logic for conditional independence. In *IEEE Symposium on Logic in Computer Science (LICS), Rome, Italy.* IEEE, 1–14.

Jialu Bao, Emanuele D'Osualdo, and Azadeh Farzan. 2024. Bluebell: An Alliance of Relational Lifting and Independence For Probabilistic Reasoning. arXiv:2402.18708

Jialu Bao, Marco Gaboardi, Justin Hsu, and Joseph Tassarotti. 2022. A separation logic for negative dependence. *Proceedings of the ACM on Programming Languages* 6, ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Philadelphia, Pennsylvania (2022), 1–29.

Gilles Barthe, Thomas Espitau, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2018. An assertion-based program logic for probabilistic programs. In *Programming Languages and Systems.* Springer International Publishing, Cham, 117–144.

Gilles Barthe, Thomas Espitau, Benjamin Grégoire, Justin Hsu, Léo Stefanesco, and Pierre-Yves Strub. 2015. Relational reasoning via probabilistic coupling. In *International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR), Suva, Fiji.* Springer, 387–401.

Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. 2009. Formal certification of code-based cryptographic proofs. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Savannah, Georgia.* 90–101.

Gilles Barthe, Justin Hsu, and Kevin Liao. 2019. A probabilistic separation logic. *Proceedings of the ACM on Programming Languages* 4, ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Lisbon, Portugal (2019), 1–30.

Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. 2012. Probabilistic relational reasoning for differential privacy. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Philadelphia, Pennsylvania*. ACM, 97–110. https://doi.org/10.1145/2103656.2103670

Nick Benton. 2004. Simple relational correctness proofs for static analyses and program transformations. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Venice, Italy*, Vol. 39. ACM New York, NY, USA, 14–25.

Richard Bornat, Cristiano Calcagno, and Hongseok Yang. 2005. Variables as resource in separation logic. In *Conference on the Mathematical Foundations of Programming Semantics (MFPS), Birmingham, England (Electronic Notes in Theoretical Computer Science, Vol. 155)*. Elsevier, 247–276.

Emanuele D'Osualdo, Azadeh Farzan, and Derek Dreyer. 2022. Proving hypersafety compositionally. *Proceedings of the ACM on Programming Languages* 6, ACM SIGPLAN Conference on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA), Auckland, New Zealand (2022), 289–314.

Michele Giry. 1982. A categorical approach to probability theory. In *Categorical Aspects of Topology and Analysis: Proceedings of an International Conference Held at Carleton University, Ottawa, August 11–15, 1981*. Springer, 68–85.

Simon Oddershede Gregersen, Alejandro Aguirre, Philipp G. Haselwarter, Joseph Tassarotti, and Lars Birkedal. 2024. Asynchronous probabilistic couplings in higher-order separation logic. *Proceedings of the ACM on Programming Languages* 8, ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), London, England (2024), 753–784.

Justin Hsu. 2017. *Probabilistic couplings for probabilistic reasoning.* Ph. D. Dissertation.

Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. 2019. The future is ours: prophecy variables in separation logic. *Proceedings of the ACM on Programming Languages* 4, ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Lisbon, Portugal (2019), 1–32.

Benjamin Lucien Kaminski. 2019. *Advanced weakest precondition calculi for probabilistic programs.* Ph. D. Dissertation. RWTH Aachen University.

Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. 2016. Weakest precondition reasoning for expected run–times of probabilistic programs. In *European Symposium on Programming (ESOP), Eindhoven, The Netherlands*. Springer, 364–389.

Dexter Kozen. 1983. A probabilistic PDL. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*. 291–297.

Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: a general, extensible modal framework for interactive proofs in separation logic. *Proceedings of the ACM on Programming Languages* 2 (2018), 77:1–77:30.

John M Li, Amal Ahmed, and Steven Holtzen. 2023a. Lilac: a modal separation logic for conditional probability. *Proceedings of the ACM on Programming Languages* 7 (2023), 148–171.

John M. Li, Amal Ahmed, and Steven Holtzen. 2023b. Lilac: A Modal Separation Logic for Conditional Probability. arXiv:2304.01339v2

John M. Li, Jon Aytac, Philip Johnson-Freyd, Amal Ahmed, and Steven Holtzen. 2024. A nominal approach to probabilistic separation logic. In *IEEE Symposium on Logic in Computer Science (LICS), Tallinn, Estonia*. ACM, 55:1–55:14.

Marcel Moosbrugger, Miroslav Stankovič, Ezio Bartocci, and Laura Kovács. 2022. This is the moment for probabilistic loops. *Proceedings of the ACM on Programming Languages* 6, ACM SIGPLAN Conference on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA), Auckland, New Zealand (2022), 1497–1525.

Carroll Morgan, Annabelle McIver, and Karen Seidel. 1996. Probabilistic predicate transformers. *ACM Transactions on Programming Languages and Systems* (1996). https://doi.org/10.1145/229542.229547

Lyle Harold Ramshaw. 1979. *Formalizing the analysis of algorithms*. Vol. 75. Xerox Palo Alto Research Center.

Robert Rand and Steve Zdancewic. 2015. VPHL: A verified partial-correctness logic for probabilistic programs. *Electronic Notes in Theoretical Computer Science* 319 (2015), 351–367.

John Reynolds. 2000. Intuitionistic reasoning about shared mutable data structure. *Millennial perspectives in computer science* 2, 1 (2000), 303–321.

Sam Staton. 2020. Probabilistic programs as measures. *Foundations of Probabilistic Programming* (2020), 43.

Joseph Tassarotti and Robert Harper. 2019. A separation logic for concurrent randomized programs. *Proceedings of the ACM on Programming Languages* 3, ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Lisbon, Portugal (2019), 1–30.

John von Neumann. 1951. Various techniques used in connection with random digits. *Journal of Research of the National Bureau of Standards, Applied Math Series* (1951), 36–38.

Yuxin Wang, Zeyu Ding, Guanhong Wang, Daniel Kifer, and Danfeng Zhang. 2019. Proving differential privacy with shadow execution. *Proceedings of the ACM on Programming Languages* ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Lisbon, Portugal (2019), 655–669.

Danfeng Zhang and Daniel Kifer. 2017. LightDP: Towards automating differential privacy proofs. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Paris, France*. 888–901.

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