Asynchronous Probabilistic Couplings in Higher-Order Separation Logic

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

Probabilistic couplings are the foundation for many probabilistic relational program logics and arise when relating random sampling statements across two programs. In relational program logics, this manifests as dedicated coupling rules that, *e.g.*, say we may reason as if two sampling statements return the same value. However, this approach fundamentally requires aligning or "synchronizing" the sampling statements of the two programs which is not always possible.

In this paper, we develop Clutch, a higher-order probabilistic relational separation logic that addresses this issue by supporting *asynchronous* probabilistic couplings. We use Clutch to develop a logical step-indexed logical relation to reason about contextual refinement and equivalence of higher-order programs written in a rich language with a probabilistic choice operator, higher-order local state, and impredicative polymorphism. Finally, we demonstrate our approach on a number of case studies.

All the results that appear in the paper have been formalized in the Coq proof assistant using the Coquelicot library and the Iris separation logic framework.

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

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

ACM Reference Format:
Simon Oddershede Gregersen, Alejandro Aguirre, Philipp G. Haselwarter, Joseph Tassarotti, and Lars Birkedal. 2024. Asynchronous Probabilistic Couplings in Higher-Order Separation Logic. *Proc. ACM Program. Lang.* 8, POPL, Article 26 (January 2024), 32 pages. https://doi.org/10.1145/3632868

1 INTRODUCTION

Relational reasoning is a useful technique for proving properties of probabilistic programs. By relating a complex probabilistic program to a simpler one, we can often reduce a challenging verification task to an easier one. In addition, certain important properties of probabilistic programs are naturally expressed in a relational form, such as stability of machine learning algorithms [Bousquet and Elisseeff 2002], differential privacy [Dwork and Roth 2013], and provable security [Goldwasser and Micali 1984]. Consequently, a number of relational program logics and models have been developed for probabilistic programs, *e.g.*, pRHL [Barthe et al. 2015], approximate pRHL

Authors' addresses: Simon Oddershede Gregersen, Aarhus University, Denmark, gregersen@cs.au.dk; Alejandro Aguirre, Aarhus University, Denmark, alejandro@cs.au.dk; Philipp G. Haselwarter, Aarhus University, Denmark, pgh@cs.au.dk; Joseph Tassarotti, New York University, USA, jt4767@cs.nyu.edu; Lars Birkedal, Aarhus University, Denmark, birke@cs.au.dk.

ACM 2475-1421/2024/1-ART26
https://doi.org/10.1145/3632868

[Barthe et al. 2016a,b, 2012], EpRHL [Barthe et al. 2018], HO-RPL [Aguirre et al. 2021], Polaris [Tassarotti and Harper 2019], logical relations [Bizjak and Birkedal 2015; Johann et al. 2010; Wand et al. 2018], and differential logical relations [Dal Lago and Gavazzo 2022].

Many probabilistic relational program logics make use of *probabilistic couplings* [Lindvall 2002; Thorisson 2000; Villani 2008], a mathematical tool for reasoning about pairs of probabilistic processes. Informally, couplings correlate outputs of two processes by specifying how corresponding sampling statements are correlated. To understand how couplings work in such logics, let us consider a pRHL-like logic. In pRHL and its variants, we prove Hoare *quadruples* of the form $\{P\}\ e_1 \sim e_2\ \{Q\}$, where e_1 and e_2 are two probabilistic programs, and P and Q are pre and post-*relations* on states of the two programs. Couplings arise when reasoning about random sampling statements in the two programs, such as in the following rule:

$$\textsc{prhl-couple}\quad \frac{}{\{P[v/x_1, v/x_2]\}\ x_1 \overset{\$}{\leftarrow} d \sim x_2 \overset{\$}{\leftarrow} d\ \{P\}}$$

Here, the two programs both sample from the same distribution d and store the result in variable x_1 and x_2, respectively. The rule says that we may reason *as if* the two sampling statements return the same value v in both programs, and one says that the sample statements have been "coupled". This is a powerful method that integrates well with existing reasoning principles from relational program logics. However, this kind of coupling rules require aligning or "synchronizing" the sampling statements of the two programs: both programs have to be executing the sample statements we want to couple for their next step when applying the rule. To enable this alignment, pRHL has various rules that enable taking steps on one side of the quadruple at a time or commuting statements in a (first-order) program. Nevertheless, with the rules from existing probabilistic relational logics, it is not always possible to synchronize sampling statements.

For example, consider the following program written in an ML-like language that *eagerly* performs a probabilistic coin flip and returns the result in a thunk:

$$\mathit{eager} \triangleq \mathsf{let}\ b = \mathsf{flip}()\ \mathsf{in}\ \lambda\_.\ b$$

An indistinguishable—but *lazy*—version of the program only does the coin flip when the thunk is invoked for the first time but stores the result in a reference that is read from in future invocations:

```
lazy ≜ let r = ref(None) in
         λ_. match !r with
               Some(b) ⇒ b
             | None    ⇒ let b = flip() in
                          r ← Some(b);
                          b
             end
```

The usual symbolic execution rules of relational logics will allow us to progress the two sides independently according to the program execution, but they will *not* allow us to line up the flip() expression in *eager* with that in *lazy*. Consequently, the coupling rule prhl-couple cannot be applied. Intuitively, the flip() expression in *eager* is evaluated immediately but the flip() expression in *lazy* only gets evaluated when the thunk is invoked—to relate the two thunks one is forced to first evaluate the eager sampling, but this then makes it impossible to couple it with the lazy sampling.

While the example may seem contrived, these kinds of transformations of eager and lazy sampling are widely used, *e.g.*, in proofs in the Random Oracle Model [Bellare and Rogaway 1993] and in game playing proofs [Bellare and Rogaway 2004, 2006]. For this reason, systems like EasyCrypt [Barthe et al. 2013] and CertiCrypt [Barthe et al. 2009, 2010] support reasoning about lazy/eager sampling through special-purpose rules for swapping statements that allows alignment of samplings; the approach is shown to work for a first-order language with global state and relies on syntactic criteria and assertions on memory disjointness. However, in rich enough languages (*e.g.* with general references and closures) these kinds of swapping-equivalences are themselves highly non-trivial, even in the non-probabilistic case [Dreyer et al. 2012; Pitts and Stark 1998].

In this paper we develop *Clutch*, a higher-order probabilistic relational separation logic that addresses this issue by enabling *asynchronous* probabilistic couplings. To do so, Clutch introduces a novel kind of ghost state, called *presampling tapes*. Presampling tapes let us reason about sampling statements as if they executed ahead of time and stored their results for later use. This converts the usual alignment problem of coupling rules into the task of reasoning about this special form of state. Fortunately, reasoning about state is well-addressed with modern separation logics.

Clutch provides a "logical" step-indexed logical relation [Dreyer et al. 2011] to reason about *contextual refinement and equivalence* of probabilistic higher-order programs written in $\mathbf{F}^{\text{rand}}_{\mu,\text{ref}}$, a rich language with a probabilistic choice operator, higher-order local state, recursive types, and impredicative polymorphism. Intuitively, expressions e_1 and e_2 of type τ are contextually equivalent if no well-typed context C can distinguish them, *i.e.*, if the expression $C[e_1]$ has the same observable behaviors as $C[e_2]$. Contextual equivalence can be decomposed into contextual refinement: we say e_1 refines e_2 at type τ, written $e_1 \precsim_{\text{ctx}} e_2 : \tau$, if, for all contexts C expecting something of type τ, if $C[e_1]$ has some observable behavior, then so does $C[e_2]$. As our language is probabilistic, here "observable behavior" means the *probability* of observing an outcome, such as termination. Using the *logical approach* [Timany et al. 2022], in Clutch, types are interpreted as relations expressed in separation logic. The resulting model allows us to prove, among other examples, that the *eager* program above is contextually equivalent to the *lazy* program.

The work presented in this paper is *foundational* [Appel 2001] in the sense that all results, including the semantics, the logic, the necessary mathematical analysis results, the relational model, and all the examples are formalized[1] in the Coq proof assistant [The Coq Development Team 2022] using the Coquelicot library [Boldo et al. 2015] and the Iris separation logic framework [Jung et al. 2016, 2018, 2015; Krebbers et al. 2017a].

In summary, we make the following contributions:

- A higher-order probabilistic relational separation logic, Clutch, for reasoning about probabilistic programs written in $\mathbf{F}^{\text{rand}}_{\mu,\text{ref}}$, an ML-like programming language with higher-order local state, recursive types, and impredicative polymorphism.
- A proof method for relating asynchronous probabilistic samplings in a program logic; a methodology that allows us to reason about sampling as if it were state and to exploit existing separation logic mechanisms such as *ghost state* and *invariants* to reason about probabilistic programs. We demonstrate the usefulness of the approach with a number of case studies.
- The first coupling-based relational program logic to reason about contextual refinement and equivalence of programs in a higher-order language with local state, recursive types, and impredicative polymorphism.
- Novel technical ideas, namely, *left-partial couplings*, a *coupling modality*, and an *erasure* argument, that allow us to prove soundness of the relational logic.
- Full mechanization in Coq using Coquelicot and the Iris separation logic framework.

[1] https://github.com/logsem/clutch

2 KEY IDEAS

The key conceptual novelties of the Clutch logic are twofold: a *logical probabilistic refinement judgment* and a novel kind of ghost resource, called *presampling tapes*.

Logical refinement. The refinement judgment $\Delta \vDash_{\mathcal{E}} e_1 \precsim e_2 : \tau$ should be read as "the expression e_1 refines the expression e_2 at type τ" and it satisfies a range of structural and symbolic execution rules as showcased in Figure 2 and further explained in §4. Just like contextual refinement, the judgment is indexed by a type τ—the environment Δ assigns semantic interpretations to type variables in τ and $\mathcal{E}$ is an *invariant mask* as elaborated on in §4. Both are safely ignored in this section. The meaning of the judgment is formally reflected by the following soundness theorem.

Theorem 1 (Soundness). *If $\emptyset \vDash e_1 \precsim e_2 : \tau$ is derivable in Clutch then $e_1 \precsim_{ctx} e_2 : \tau$.*

The refinement judgment is *internal* to the ambient Clutch separation logic. This means that we can combine the judgment in arbitrary ways with other logical connectives: *e.g.*, the *separating conjunction* $P * Q$ and its adjoint *separating implication* (magic wand) $P \mathrel{-\!\!*} Q$. All inference rules that we present can be internalized as propositions in the logic and we will use an inference rule with premises $P_1, \ldots, P_n$ and conclusion Q as notation for $(P_1 * \ldots * P_n) \vdash Q$.

The language $\mathbf{F}^{\text{rand}}_{\mu,\text{ref}}$ contains a single probabilistic primitive $\text{rand}(N)$ that reduces uniformly at random to some $n \in \{0, 1, \ldots, N\}$:

$$\text{rand}(N), \sigma \rightarrow^{1/(N+1)} n, \sigma \qquad\qquad n \in \{0, 1, \ldots, N\}$$

where σ is the current program state and $\rightarrow \subseteq Cfg \times [0, 1] \times Cfg$ is a small-step transition relation, annotated with the probability that the transition occurs. By defining $\text{flip}() \triangleq \text{if } \text{rand}(1) = 0 \text{ then false else true}$ we recover the Boolean fair coin flip operator used in the motivating example. To reason relationally about probabilistic choices that *can* be synchronized, Clutch admits a classical coupling rule that allows us to continue reasoning as if the two sampled values are related by a bijection f on the sampling space $\{0, \ldots, N\}$:

$$\textsc{rel-couple-rands}\quad \frac{f \text{ bijection} \qquad \forall n \leq N.\, \Delta \vDash_{\mathcal{E}} K[n] \precsim K'[f(n)] : \tau}{\Delta \vDash_{\mathcal{E}} K[\,\text{rand}(N)] \precsim K'[\,\text{rand}(N)] : \tau}$$

where K and K' are arbitrary evaluation contexts.

Asynchronous couplings. To support *asynchronous* couplings we introduce *presampling tapes*. Reminiscent of how *prophecy variables* [Abadi and Lamport 1988, 1991; Jung et al. 2020] allow us to talk about the future, presampling tapes give us the means to talk about the outcome of probabilistic choices *in the future*.[2] Tapes manifest both in the operational semantics and in the logic.

Operationally, a tape consists of an upper bound $N \in \mathbb{N}$ and a finite sequence of natural numbers less than or equal to N, representing future outcomes of $\text{rand}(N)$ commands. Each tape is labeled with an identifier $\iota \in Label$, and a program's state is extended with a finite map from labels to tapes. Tapes can be dynamically allocated using a tape primitive:

$$\text{tape}(N), \sigma \rightarrow^{1} \iota, \sigma[\iota \mapsto (N, \epsilon)] \qquad\qquad \text{if } \iota = \text{fresh}(\sigma)$$

which extends the mapping with an empty tape and the upper bound N, and it returns its fresh label ι. The rand primitive can then optionally be annotated with a tape label ι. If $\sigma(\iota) = (N, \epsilon)$, *i.e.*, the corresponding tape is empty, $\text{rand}(N, \iota)$ reduces to any $n \leq N$ with equal probability:

$$\text{rand}(N, \iota), \sigma \rightarrow^{1/(N+1)} n, \sigma \qquad\qquad \text{if } \sigma(\iota) = (N, \epsilon) \text{ and } n \leq N$$

[2] As showcased in §7, however, prophecy variables as previously developed in Iris are unsound for the coupling- logic.

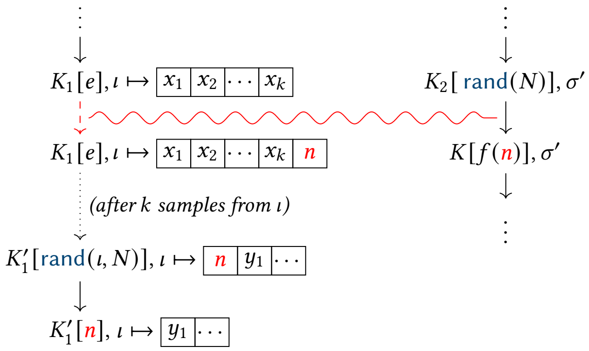
$K_1[e], \iota \mapsto \boxed{x_1}\boxed{x_2}\boxed{\cdots}\boxed{x_k}$
$K_2[\,\mathsf{rand}(N)], \sigma'$
$K_1[e], \iota \mapsto \boxed{x_1}\boxed{x_2}\boxed{\cdots}\boxed{x_k}\boxed{n}$
$K[f(n)], \sigma'$
(after k samples from ι)
$K_1'[\mathsf{rand}(\iota, N)], \iota \mapsto \boxed{n}\boxed{y_1}\boxed{\cdots}$
$K_1'[n], \iota \mapsto \boxed{y_1}\boxed{\cdots}$

Fig. 1. Illustration of an asynchronous coupling established through the rule REL-COUPLE-TAPE-L.

but if the tape is *not* empty, the $\mathsf{rand}(N, \iota)$ primitive reduces *deterministically* by taking off the first element of the tape and returning it:

$$\mathsf{rand}(N, \iota), \sigma \to^1 n, \sigma[\iota \mapsto (N, \vec{n})\,] \qquad\qquad \text{if } \sigma(\iota) = (N, n \cdot \vec{n})$$

If the tape bounds do not match, then $\mathsf{rand}(N, \iota)$ reduces as if the tape was empty:

$$\mathsf{rand}(N, \iota), \sigma \to^{1/(N+1)} n, \sigma \qquad\qquad \text{if } \sigma(\iota) = (M, \vec{n}) \text{ and } N \neq M \text{ and } n \leq N$$

However, *no* primitives in the language add values to the tapes! Instead, values are added to tapes as part of presampling steps that will be *ghost operations* appearing only in the relational logic. That is, presampling will purely be a proof-device that has no operational effect: in the end, tapes can in fact be erased entirely through refinement as will be clear by the end of this section.

At the logical level, Clutch comes with a $\iota \hookrightarrow (N, \vec{n})$ assertion that denotes *ownership* of the label ι and its contents $(N, \vec{n})$, analogously to how the traditional points-to-connective $\ell \mapsto v$ of separation logic denotes ownership of the location ℓ and its contents on the heap. When a tape is allocated, ownership of the fresh empty tape is acquired, *i.e.*,

$$\frac{\forall \iota.\, \iota \hookrightarrow (N, \epsilon) -\!\!* \Delta \vDash_{\mathcal{E}} K[\iota] \precsim e : \tau}{\Delta \vDash_{\mathcal{E}} K[\,\mathsf{tape}(N)] \precsim e : \tau}\ \textsc{rel-alloc-tape-l}$$

Asynchronous couplings between probabilistic choices can be established in the refinement logic by coupling ghost presamplings with program steps. For example, the rule below allows us to couple an (unlabeled) probabilistic choice on the right with a presampling on the ι tape on the left:

$$\frac{f \text{ bijection} \qquad e \notin \mathit{Val} \qquad \iota \hookrightarrow (N, \vec{n}) \qquad \forall n \leq N.\, \iota \hookrightarrow (N, \vec{n} \cdot n) -\!\!* \Delta \vDash_{\mathcal{E}} e \precsim K'[f(n)] : \tau}{\Delta \vDash_{\mathcal{E}} e \precsim K'[\,\mathsf{rand}(N)] : \tau}\ \textsc{rel-couple-tape-l}$$

Intuitively, as illustrated in Figure 1, the rule allows us to couple a logical *ghost* presampling step on the left (illustrated using a red dashed arrow) with a *physical* sampling on the right. A symmetric rule holds for the opposite direction and two ghost presamplings can be coupled as well. When we—at some point in the future—reach a presampled $\mathsf{rand}(N, \iota)$, we simply read off the presampled values from the ι tape deterministically in a first-in-first-out order, *i.e.*,

$$\frac{\iota \hookrightarrow (N, n \cdot \vec{n}) \qquad \iota \hookrightarrow (N, \vec{n}) -\!\!* \Delta \vDash_{\mathcal{E}} K[n] \precsim e_2 : \tau}{\Delta \vDash_{\mathcal{E}} K[\,\mathsf{rand}(N, \iota)] \precsim e_2 : \tau}\ \textsc{rel-rand-tape-l}$$

If we do not perform any presamplings, tapes and labels can be ignored and we can couple labeled sampling commands as if they were unlabeled:

$$\textsc{rel-rand-erase-r}\quad \frac{\iota \hookrightarrow_{\mathsf{s}} (N, \epsilon) \qquad \forall n \leq N.\, \Delta \vDash_{\mathcal{E}} K[n] \precsim K'[n] : \tau}{\Delta \vDash_{\mathcal{E}} K[\,\mathsf{rand}(N)] \precsim K'[\,\mathsf{rand}(N, \iota)] : \tau}$$

Here the assertion $\iota \hookrightarrow_{\mathsf{s}} (N, \epsilon)$ denotes ownership of an empty tape ι of the right-hand side program (the program on the "specification" side).

Example. Using presampling tapes, we can show that *lazy* is a contextual refinement of *eager* from §1, that is, $\mathit{lazy} \precsim_{\mathrm{ctx}} \mathit{eager} : \mathsf{unit} \to \mathsf{bool}$. We first define an intermediate labeled version of *lazy*, using $\mathsf{flip}(\iota) \triangleq \mathsf{if}\ \mathsf{rand}(1, \iota) = 0\ \mathsf{then\ false\ else\ true}$:

$$\begin{aligned} \mathit{lazy}' \triangleq\ & \mathsf{let}\ \iota = \mathsf{tape}(1)\ \mathsf{in} \\ & \mathsf{let}\ r = \mathsf{ref}(\mathsf{None})\ \mathsf{in} \\ & \lambda\_.\ \mathsf{match}\ !\,r\ \mathsf{with} \\ & \qquad \mathsf{Some}(b) \Rightarrow b \\ & \qquad |\ \mathsf{None} \quad\ \Rightarrow \mathsf{let}\ b = \mathsf{flip}(\iota)\ \mathsf{in} \\ & \qquad\qquad\qquad\quad\ r \leftarrow \mathsf{Some}(b); \\ & \qquad\qquad\qquad\quad\ b \\ & \quad\ \mathsf{end} \end{aligned}$$

By transitivity of contextual refinement and Theorem 1 it suffices to show $\vDash \mathit{lazy} \precsim \mathit{lazy}' : \mathsf{unit} \to \mathsf{bool}$ and $\vDash \mathit{lazy}' \precsim \mathit{eager} : \mathsf{unit} \to \mathsf{bool}$. The former follows straightforwardly using symbolic execution rules and rel-rand-erase-r. To show the latter we allocate a tape ι and a reference ℓ on the left by symbolic execution and couple the presampling of a $b \in \{0, 1\}$ on the ι tape with the $\mathsf{flip}()$ on the right using rel-couple-tape-l. This establishes an *invariant*

$$(\iota \hookrightarrow (1, b) * \ell \mapsto \mathsf{None}) \vee \ell \mapsto \mathsf{Some}(b)$$

that expresses how either b is on the ι tape and the location ℓ is empty *or* ℓ contains the value b. Invariants are particular kinds of propositions in Clutch that, in this particular case, are guaranteed to always hold at the beginning and at the end of the function evaluation. Under this invariant, we show that the two thunks are related by symbolic execution and rules for accessing invariants that we detail in §4. Symmetric arguments allow us to show the refinement in the other direction and consequently the contextual equivalence.

This example shows how presampling tapes are simple and powerful, yet merely a proof-device: the final equivalence holds for programs without any mention of tapes. Intuitively, tapes allow us to separate the process of building a coupling from the operational semantics of the program. One might be tempted to believe, though, that as soon as the idea of presampling arises, the high-level proof rules as supported by Clutch are straightforward to state and prove. This is *not* the case. As we will show throughout the paper, a great deal of care goes into defining a system that supports presampling while being sound. In §7 we discuss two counterexamples that illustrate some of the subtleties involved in defining a sound system.

3 PRELIMINARIES AND THE LANGUAGE $\mathbf{F}^{\mathrm{rand}}_{\mu,\mathrm{ref}}$

To account for non-terminating behavior, we will define our operational semantics using probability sub-distributions which we recall below.

Definition 2 (Sub-distribution). *A (discrete)* sub-distribution *over a countable set A is a function $\mu : A \to [0, 1]$ such that $\sum_{a \in A} \mu(a) \leq 1$. We write $\mathcal{D}(A)$ for the set of all sub-distributions over A.*

Definition 3 (Support). *The* support *of $\mu \in \mathcal{D}(A)$ is the set of elements*

$$\mathrm{supp}(\mu) \triangleq \{a \in A \mid \mu(a) > 0\}$$

Lemma 4 (Probability Monad). *Let $\mu \in \mathcal{D}(A)$, $a \in A$, and $f : A \to \mathcal{D}(B)$. Then*

(1) $\mathrm{bind}(f, \mu)(b) \triangleq \sum_{a \in A} \mu(a) \cdot f(a)(b)$

(2) $\mathrm{ret}(a)(a') \triangleq \begin{cases} 1 & \textit{if } a = a' \\ 0 & \textit{otherwise} \end{cases}$

gives monadic structure to $\mathcal{D}$. We write $\mu \mathrel{>\!\!>\!=} f$ for $\mathrm{bind}(f, \mu)$.

The syntax of the language $\mathbf{F}^{\text{rand}}_{\mu,\text{ref}}$ is defined by the grammar below.

$$\begin{aligned}
v, w \in \mathit{Val} &::= z \in \mathbb{Z} \mid b \in \mathbb{B} \mid () \mid \ell \in \mathit{Loc} \mid \iota \in \mathit{Label} \mid \mathsf{rec}\, f\ x = e \mid (v, w) \mid \mathsf{inl}(v) \mid \mathsf{inr}(v) \\
e \in \mathit{Expr} &::= v \mid x \mid e_1(e_2) \mid \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid \mathsf{fst}(e) \mid \mathsf{snd}(e) \mid \mathsf{ref}(e) \mid\ !\, e \mid e_1 \leftarrow e_2 \mid \\
&\qquad \mathsf{match}\ e\ \mathsf{with}\ \mathsf{inl}(v) \Rightarrow e_1 \mid \mathsf{inr}(w) \Rightarrow e_2\ \mathsf{end} \mid \mathsf{fold}\ e \mid \mathsf{unfold}\ e \mid \Lambda e \mid e\ \_ \mid \\
&\qquad \mathsf{pack}\ e \mid \mathsf{unpack}\ e\ \mathsf{as}\ x\ \mathsf{in}\ e \mid \mathsf{tape}(e) \mid \mathsf{rand}(e_1, e_2) \mid e_1 + e_2 \mid e_1 - e_2 \mid \cdots \\
K \in \mathit{Ectx} &::= - \mid e\, K \mid K\, v \mid\ !\, K \mid e \leftarrow K \mid K \leftarrow v \mid \mathsf{tape}(K) \mid \mathsf{rand}(e, K) \mid \mathsf{rand}(K, v) \mid \ldots \\
\sigma \in \mathit{State} &\triangleq (\mathit{Loc} \xrightharpoonup{\text{fin}} \mathit{Val}) \times (\mathit{Label} \xrightharpoonup{\text{fin}} \mathit{Tape}) \\
t \in \mathit{Tape} &\triangleq \{(N, \vec{n}) \mid N \in \mathbb{N} \land \vec{n} \in \mathbb{N}^*_{\leq N}\} \\
\rho \in \mathit{Cfg} &\triangleq \mathit{Expr} \times \mathit{State} \\
\tau \in \mathit{Type} &::= \alpha \mid \mathsf{unit} \mid \mathsf{bool} \mid \mathsf{nat} \mid \mathsf{int} \mid \tau \times \tau \mid \tau + \tau \mid \tau \to \tau \mid \forall \alpha.\, \tau \mid \exists \alpha.\, \tau \mid \mu\, \alpha.\, \tau \mid \mathsf{ref}\ \tau \mid \mathsf{tape}
\end{aligned}$$

The term language is mostly standard but note that there are no types in terms; we write Λe for type abstraction and $e\ \_$ for type application. $\mathsf{fold}\ e$ and $\mathsf{unfold}\ e$ are the special term constructs for iso-recursive types. $\mathsf{ref}(e)$ allocates a new reference, $!\, e$ dereferences the location e evaluates to, and $e_1 \leftarrow e_2$ assigns the result of evaluating e_2 to the location that e_1 evaluates to. We introduce syntactic sugar for lambda abstractions $\lambda x.\, e$ defined as $\mathsf{rec}\ \_\ x = e$, let-bindings $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ defined as $(\lambda x.\, e_2)(e_1)$, and sequencing $e_1; e_2$ defined as $\mathsf{let}\ \_ = e_1\ \mathsf{in}\ e_2$. We write $\mathsf{rand}(N)$ for $\mathsf{rand}(N, ())$, *i.e.* an unlabeled probabilistic choice.

We implicitly coerce from $\sigma \in \mathit{State}$ to heaps and tapes, *e.g.*, $\sigma(\ell) = \pi_1(\sigma)(\ell)$ and $\sigma(\iota) = \pi_2(\sigma)(\iota)$. Tapes are formally pairs $(N, \vec{n})$ of $N \in \mathbb{N}$ and a finite sequence $\vec{n}$ of natural numbers less than or equal to N. The language has a call-by-value single-step-reduction relation $\to \subseteq \mathit{Cfg} \times [0, 1] \times \mathit{Cfg}$ defined using evaluation contexts $K \in \mathit{Ectx}$. The relation is mostly standard: all the non-probabilistic constructs reduce as usual with weight 1 and $\mathsf{rand}(e_1, e_2)$ reduces as discussed in §2.

To define full program execution, let $\mathrm{step}(\rho) \in \mathcal{D}(\mathit{Cfg})$ denote the distribution induced by the single step reduction of configuration $\rho \in \mathit{Cfg}$. First, we define a stratified execution probability $\mathrm{exec}_n \colon \mathit{Cfg} \to \mathcal{D}(\mathit{Val})$ by induction on n:

$$\mathrm{exec}_n(e, \sigma) \triangleq \begin{cases} \mathbf{0} & \text{if } e \notin \mathit{Val} \text{ and } n = 0 \\ \mathrm{ret}(e) & \text{if } e \in \mathit{Val} \\ \mathrm{step}(e, \sigma) \mathrel{>\!\!>\!=} \mathrm{exec}_{(n-1)} & \text{otherwise} \end{cases}$$

where $\mathbf{0}$ denotes the everywhere-zero distribution. That is, $\mathrm{exec}_n(e, \sigma)(v)$ denotes the probability of stepping from the configuration (e, σ) to a value v in less than n steps. The probability that a full execution, starting from configuration ρ, reaches a value v is the limit of its stratified approximations, which exists by monotonicity and boundedness:

$$\mathrm{exec}(\rho)(v) \triangleq \lim_{n \to \infty} \mathrm{exec}_n(\rho)(v)$$

The probability that a full execution from a starting configuration ρ terminates then becomes $\mathrm{exec}_{\Downarrow}(\rho) \triangleq \sum_{v \in \mathit{Val}} \mathrm{exec}(\rho)(v)$.

Typing judgments have the form $\Theta \mid \Gamma \vdash e : \tau$ where Γ is a context assigning types to program variables, and Θ is a context of type variables that may occur in Γ and τ. The inference rules for the typing judgments are standard (see, *e.g.*, Frumin et al. [2021b] or the Coq formalization) and omitted, except for the straightforward rules for typing tapes and samplings shown below:

$$\textsc{T-Tape}\ \frac{\Theta \mid \Gamma \vdash e : \mathsf{nat}}{\Theta \mid \Gamma \vdash \mathsf{tape}(e) : \mathsf{tape}} \qquad \textsc{T-Rand}\ \frac{\Theta \mid \Gamma \vdash e_1 : \mathsf{nat} \qquad \Theta \mid \Gamma \vdash e_2 : \tau \qquad \tau = \mathsf{unit} \lor \tau = \mathsf{tape}}{\Theta \mid \Gamma \vdash \mathsf{rand}(e_1, e_2) : \mathsf{nat}}$$

The notion of contextual refinement that we use is also mostly standard and uses the termination probability $\mathrm{exec}_{\Downarrow}$ as observation predicate. Since we are in a typed setting, we consider only typed contexts. A program context is well-typed, written $C : (\Theta \mid \Gamma \vdash \tau) \Rightarrow (\Theta' \mid \Gamma' \vdash \tau')$, if for any term e such that $\Theta \mid \Gamma \vdash e : \tau$ we have $\Theta' \mid \Gamma' \vdash C[e] : \tau'$. We say expression e_1 *contextually refines* expression e_2 if for all well-typed program contexts C resulting in a closed program then the termination probability of $C[e_1]$ is bounded by the termination probability of $C[e_2]$:

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

Note that contextual refinement is a precongruence, and that the statement itself is in the meta-logic (*e.g.*, Coq) and makes no mention of Clutch or Iris. Contextual equivalence $\Theta \mid \Gamma \vdash e_1 \simeq_{\mathrm{ctx}} e_2 : \tau$ is defined as the symmetric interior of refinement: $(\Theta \mid \Gamma \vdash e_1 \precsim_{\mathrm{ctx}} e_2 : \tau) \land (\Theta \mid \Gamma \vdash e_2 \precsim_{\mathrm{ctx}} e_1 : \tau)$.

4 THE CLUTCH REFINEMENT LOGIC

In the style of ReLoC [Frumin et al. 2021b], we define a *logical refinement judgment* $\Delta \vDash_{\mathcal{E}} e_1 \precsim e_2 : \tau$ as an internal notion in the Clutch separation logic by structural recursion over the type τ. The fundamental theorem of logical relations will then show that logical refinement implies contextual refinement. This means proving contextual refinement can be reduced to proving logical refinement, which is generally much easier. When defining and proving logical refinement, we can leverage the features of modern separation logic, *e.g.*, (impredicative) invariants and (higher-order) ghost state as inherited from Iris, to model and reason about complex programs and language features.

Clutch is based on higher-order intuitionistic separation logic and the most important propositions are shown below.

$$P, Q \in \mathit{iProp} ::= \mathsf{True} \mid \mathsf{False} \mid P \land Q \mid P \lor Q \mid P \Rightarrow Q \mid \forall x.\, P \mid \exists x.\, P \mid P * Q \mid P \mathrel{-\!\!*} Q \mid \\ \square\, P \mid \triangleright P \mid \mu x.\, P \mid \ulcorner \phi \urcorner \mid \boxed{P}^{\mathcal{N}} \mid \boxed{P}^{\mathcal{N}} \mid \ell \mapsto v \mid \ell \mapsto_{\mathsf{s}} v \mid \\ \iota \hookrightarrow (N, \vec{n}) \mid \iota \hookrightarrow_{\mathsf{s}} (N, \vec{n}) \mid [\![\tau]\!]_{\Delta}(v_1, v_2) \mid \Delta \vDash_{\mathcal{E}} e_1 \precsim e_2 : \tau \mid \ldots$$

As Clutch is built upon the base logic of Iris [Jung et al. 2018], it includes all its connectives such as the *persistence* modality $\square$, the *later* modality $\triangleright$, fixpoints $\mu x.\, P$, *invariants* $\boxed{P}^{\mathcal{N}}$, and *non-atomic invariants* [The Iris Development Team 2022], written $\boxed{P}^{\mathcal{N}}$, which we will introduce as needed. The proposition $\ulcorner \phi \urcorner$ embeds a meta-logic (*e.g.*, Coq) proposition ϕ (*e.g.*, equality or a coupling) into Clutch but we will omit the brackets whenever the type of ϕ is clear from the context.

Like ordinary separation logic, Clutch has heap points-to assertions. Since the logic is relational, these come in two forms: $\ell \mapsto v$ for the left-hand side program's state and $\ell \mapsto_{\mathsf{s}} v$ for the right-hand side's state (the "specification" side). For the same reason, tape assertions come in two forms as well, $\iota \hookrightarrow (N, \vec{n})$ and $\iota \hookrightarrow_{\mathsf{s}} (N, \vec{n})$ respectively.

4.1 Refinement Judgments

The refinement judgment $\Delta \vDash_{\mathcal{E}} e_1 \precsim e_2 : \tau$ should be read as "in environment Δ, the expression e_1 refines the expression e_2 at type τ under the invariants in $\mathcal{E}$". We refer to e_1 as the *implementation* and to e_2 as the *specification*. The environment Δ assigns *interpretations* to type variables occurring in τ. These interpretations are Clutch relations of type $\mathit{Val} \times \mathit{Val} \to \mathit{iProp}$. One such relation is the binary interpretation $[\![\tau]\!]_\Delta(-,-)$ of a syntactic type $\tau \in \mathit{Type}$ which is used to define the refinement judgment, as discussed in §5.2. For example, for base types such as bool and int, the value interpretation asserts equality between the values.

Figure 2 showcases a selection of the type-directed structural and computational rules for proving logical refinement for deterministic reductions. Our computational rules resemble the typical forward-symbolic-execution-style rules from, *e.g.*, the weakest precondition calculus in Iris [Jung et al. 2018], but come in forms for both the left-hand side and the right-hand side. For example, REL-PURE-L and REL-PURE-R symbolically execute "pure" reductions, *i.e.* reductions that do not depend on the state, such as β-reductions. REL-STORE-L and REL-STORE-R on the other hand depend on the heap and require ownership of a location to store values at it. We remark that all the rules for the deterministic fragment of the Clutch refinement judgment are identical to the rules for the sequential fragment of the non-probabilistic relational logic ReLoC [Frumin et al. 2021b]—even though the underlying semantics and model are very different. This is one of the key reasons behind the support for modular reasoning.

The rules in Figure 3 showcase the computational rules for non-coupled probabilistic reductions and for interactions with presampling tapes. The rules REL-RAND-TAPE-L and REL-RAND-TAPE-R allow us to read off values from a tape as explained in §2; if the tapes are empty, REL-RAND-TAPE-EMPTY-L and REL-RAND-TAPE-EMPTY-R continue with a fresh sampling just like for unlabeled rands in REL-RAND-L and REL-RAND-R. Notice how the rules resemble the rules for interacting with the heap.

The main novelty of Clutch is the support for both synchronous and asynchronous couplings for which rules are shown in Figure 4. REL-COUPLE-RANDS is a classical coupling rule that relates two samplings that can be aligned, just like PRHL-COUPLE as we saw in §1. The rules REL-COUPLE-TAPE-L and REL-COUPLE-TAPE-R, on the other hand, are asynchronous coupling rules; they both couple a sampling reduction with an arbitrary expression on the opposite side by presampling a coupled value to a tape, as discussed in §2. Finally, REL-COUPLE-TAPES couples two ghost presamplings to two tapes, and hence offers full asynchrony.

4.2 Persistence and Invariants

As mentioned above, the environment Δ in Clutch's refinement judgement provides an interpretation of types as relations in the logic. However, Clutch is a substructural separation logic, while the type system of $\mathbf{F}^{\text{rand}}_{\mu,\text{ref}}$ is *not* substructural. To account for the non-substructural nature of $\mathbf{F}^{\text{rand}}_{\mu,\text{ref}}$'s types, we make use of the *persistence modality* $\square$. We say P is persistent, written $\text{persistent}(P)$ if $P \vdash \square P$; otherwise, we say that P is *ephemeral*. Persistent resources can freely be duplicated ($\square P \dashv\vdash \square P * \square P$) and eliminated ($\square P \vdash P$). For example, invariants and non-atomic invariants are persistent: once established, they will remain true forever. On the contrary, ephemeral propositions like the points-to connective $\ell \mapsto v$ for the heap may be invalidated in the future when the location is updated. For exactly this reason, the rule REL-PACK also requires the interpretation of the type variable to be persistent, to guarantee that it does not depend on ephemeral resources.

To reason about, *e.g.*, functions that make use of ephemeral resources, a common pattern is to "put them in an invariant" to make them persistent, as sketched in §2 for the lazy/eager example. Since our language is sequential, when a function is invoked, no other code can execute before the function returns. This means that we can soundly keep invariants "open" and temporarily

REL-PURE-L
$$\frac{e_1 \overset{\text{pure}}{\rightsquigarrow} e_1' \qquad \Delta \vDash_{\mathcal{E}} K[e_1'] \precsim e_2 : \tau}{\Delta \vDash_{\mathcal{E}} K[e_1] \precsim e_2 : \tau}$$

REL-PURE-R
$$\frac{e_2 \overset{\text{pure}}{\rightsquigarrow} e_2' \qquad \Delta \vDash_{\mathcal{E}} e_1 \precsim K[e_2'] : \tau}{\Delta \vDash_{\mathcal{E}} e_1 \precsim K[e_2] : \tau}$$

REL-ALLOC-L
$$\frac{\forall \ell.\, \ell \mapsto v \twoheadrightarrow \Delta \vDash_{\mathcal{E}} K[\ell] \precsim e_2 : \tau}{\Delta \vDash_{\mathcal{E}} K[\mathsf{ref}(v)] \precsim e_2 : \tau}$$

REL-ALLOC-R
$$\frac{\forall \ell.\, \ell \mapsto_{\mathsf{s}} v \twoheadrightarrow \Delta \vDash_{\mathcal{E}} e_1 \precsim K[\ell] : \tau}{\Delta \vDash_{\mathcal{E}} e_1 \precsim K[\mathsf{ref}(v)] : \tau}$$

REL-LOAD-L
$$\frac{\ell \mapsto v \qquad \ell \mapsto v \twoheadrightarrow \Delta \vDash_{\mathcal{E}} K[v] \precsim e_2 : \tau}{\Delta \vDash_{\mathcal{E}} K[!\ell] \precsim e_2 : \tau}$$

REL-LOAD-R
$$\frac{\ell \mapsto_{\mathsf{s}} v \qquad \ell \mapsto_{\mathsf{s}} v \twoheadrightarrow \Delta \vDash_{\mathcal{E}} e_1 \precsim K[v] : \tau}{\Delta \vDash_{\mathcal{E}} e_1 \precsim K[!\ell] : \tau}$$

REL-STORE-L
$$\frac{\ell \mapsto v \qquad \ell \mapsto w \twoheadrightarrow \Delta \vDash_{\mathcal{E}} K[()] \precsim e_2 : \tau}{\Delta \vDash_{\mathcal{E}} K[\ell \leftarrow w] \precsim e_2 : \tau}$$

REL-STORE-R
$$\frac{\ell \mapsto_{\mathsf{s}} v \qquad \ell \mapsto_{\mathsf{s}} w \twoheadrightarrow \Delta \vDash_{\mathcal{E}} e_1 \precsim K[()] : \tau}{\Delta \vDash_{\mathcal{E}} e_1 \precsim K[\ell \leftarrow w] : \tau}$$

REL-PACK
$$\frac{\forall v_1, v_2.\, \mathsf{persistent}(R(v_1, v_2)) \qquad \Delta, \alpha \mapsto R \vDash_{\top} e_1 \precsim e_2 : \tau}{\Delta \vDash_{\top} \mathsf{pack}\, e_1 \precsim \mathsf{pack}\, e_2 : \exists \alpha.\, \tau}$$

REL-REC
$$\frac{\Box\left(\forall v_1, v_2.\, [\![\tau]\!]_{\Delta}(v_1, v_2) \twoheadrightarrow \Delta \vDash_{\top} (\mathsf{rec}\, f_1\, x_1 = e_1)\, v_1 \precsim (\mathsf{rec}\, f_2\, x_2 = e_2)\, v_2 : \tau \to \sigma\right)}{\Delta \vDash_{\top} \mathsf{rec}\, f_1\, x_1 = e_1 \precsim \mathsf{rec}\, f_2\, x_2 = e_2 : \tau \to \sigma}$$

REL-RETURN
$$\frac{[\![\tau]\!]_{\Delta}(v_1, v_2)}{\Delta \vDash_{\top} v_1 \precsim v_2 : \tau}$$

REL-BIND
$$\frac{\Delta \vDash_{\mathcal{E}} e_1 \precsim e_2 : \tau \qquad \forall v_1, v_2.\, [\![\tau]\!]_{\Delta}(v_1, v_2) \twoheadrightarrow \Delta \vDash_{\top} K[v_1] \precsim K'[v_2] : \sigma}{\Delta \vDash_{\mathcal{E}} K[e_1] \precsim K'[e_2] : \sigma}$$

Fig. 2. Selected structural and symbolic execution rules for the Clutch refinement judgment.

invalidate them for the entire duration of a function invocation—as long as the invariants are reestablished before returning. Non-atomic invariants allow us to capture exactly this intuition.

Invariants are annotated with invariant names $\mathcal{N} \in \mathit{InvName}$ and the refinement judgment is annotated by *invariant masks* $\mathcal{E} \subseteq \mathit{InvName}$ that indicates which non-atomic invariants that are currently *closed*. This is needed for bookkeeping of the invariant mechanism in order to avoid reentrancy issues, where invariants are opened in a nested (and unsound) fashion.

Figure 5 shows structural rules for the refinement judgment's interaction with non-atomic invariants. An invariant $\boxed{P}^{\mathcal{N}}$ can be allocated (rel-na-inv-alloc) by giving up ownership of P. When opening an invariant (rel-na-inv-open) one obtains the resources P together with a resource $\mathsf{closeNaInv}_{\mathcal{N}}(P)$ that allows one to close the invariant again (rel-na-inv-close) by reestablishing P. We guarantee that all invariants are closed by the end of evaluation by requiring $\top$, the set of all invariant names, as mask annotation on the judgment in all value cases (see, *e.g.*, rel-rec, rel-pack, and rel-return in Figure 2).

Clutch invariants are inherited from Iris and hence they are *impredicative* [Svendsen and Birkedal 2014] which means that the proposition P in $\boxed{P}^{\mathcal{N}}$ is *arbitrary* and can, *e.g.*, contain other invariant assertions. To ensure soundness of the logic and avoid self-referential paradoxes, invariant access guards P by the later modality $\triangleright$. When invariants are not used impredicatively, the later modality

REL-RAND-L

$$\frac{\forall n \leq N.\, \Delta \vDash_{\mathcal{E}} K[n] \precsim e_2 : \tau}{\Delta \vDash_{\mathcal{E}} K[\,\mathsf{rand}(N)] \precsim e_2 : \tau}$$

REL-RAND-R

$$\frac{e_1 \notin \mathit{Val} \qquad \forall n \leq N.\, \Delta \vDash_{\mathcal{E}} e_1 \precsim K[b] : \tau}{\Delta \vDash_{\mathcal{E}} e_1 \precsim K[\,\mathsf{rand}(N)] : \tau}$$

REL-ALLOC-TAPE-L

$$\frac{\forall \iota.\, \iota \hookrightarrow (N, \epsilon) \mathrel{-\!\!*} \Delta \vDash K[\iota] \precsim e : \tau}{\Delta \vDash K[\,\mathsf{tape}(N)] \precsim e : \tau}$$

REL-ALLOC-TAPE-R

$$\frac{\forall \iota.\, \iota \hookrightarrow_{\mathsf{s}} (N, \epsilon) \mathrel{-\!\!*} \Delta \vDash e \precsim K[\iota] : \tau}{\Delta \vDash e \precsim K[\,\mathsf{tape}(N)] : \tau}$$

REL-RAND-TAPE-L

$$\frac{\iota \hookrightarrow (N, n \cdot \vec{n}) \qquad \iota \hookrightarrow (N, \vec{n}) \mathrel{-\!\!*} \Delta \vDash_{\mathcal{E}} K[n] \precsim e_2 : \tau}{\Delta \vDash_{\mathcal{E}} K[\,\mathsf{rand}(N, \iota)] \precsim e_2 : \tau}$$

REL-RAND-TAPE-R

$$\frac{\iota \hookrightarrow_{\mathsf{s}} (N, n \cdot \vec{n}) \qquad \iota \hookrightarrow_{\mathsf{s}} (N, \vec{n}) \mathrel{-\!\!*} \Delta \vDash_{\mathcal{E}} e_1 \precsim K[n] : \tau}{\Delta \vDash_{\mathcal{E}} e_1 \precsim K[\,\mathsf{rand}(N, \iota)] : \tau}$$

REL-RAND-TAPE-EMPTY-L

$$\frac{\iota \hookrightarrow (N, \epsilon) \qquad \forall n \leq N.\, \iota \hookrightarrow (N, \epsilon) \mathrel{-\!\!*} \Delta \vDash_{\mathcal{E}} K[n] \precsim e_2 : \tau}{\Delta \vDash_{\mathcal{E}} K[\,\mathsf{rand}(N, \iota)] \precsim e_2 : \tau}$$

REL-RAND-TAPE-EMPTY-R

$$\frac{e_1 \notin \mathit{Val} \qquad \iota \hookrightarrow_{\mathsf{s}} (N, \epsilon) \qquad \forall n \leq N.\, \iota \hookrightarrow_{\mathsf{s}} (N, \epsilon) \mathrel{-\!\!*} \Delta \vDash_{\mathcal{E}} e_1 \precsim K[n] : \tau}{\Delta \vDash_{\mathcal{E}} e_1 \precsim K[\,\mathsf{rand}(N, \iota)] : \tau}$$

Fig. 3. Rules for non-relational probabilistic choices and tapes for the Clutch refinement judgment.

REL-COUPLE-RANDS

$$\frac{f \text{ bijection} \qquad \forall n \leq N.\, \Delta \vDash_{\mathcal{E}} K[n] \precsim K'[f(n)] : \tau}{\Delta \vDash_{\mathcal{E}} K[\,\mathsf{rand}(N)] \precsim K'[\,\mathsf{rand}(N)] : \tau}$$

REL-COUPLE-TAPE-L

$$\frac{f \text{ bijection} \qquad e_1 \notin \mathit{Val} \qquad \iota \hookrightarrow (N, \vec{n}) \qquad \forall n \leq N.\, \iota \hookrightarrow (N, \vec{n} \cdot n) \mathrel{-\!\!*} \Delta \vDash_{\mathcal{E}} e_1 \precsim K[f(n)] : \tau}{\Delta \vDash_{\mathcal{E}} e_1 \precsim K[\,\mathsf{rand}(N)] : \tau}$$

REL-COUPLE-TAPE-R

$$\frac{f \text{ bijection} \qquad \iota \hookrightarrow_{\mathsf{s}} (N, \vec{n}) \qquad \forall n \leq N.\, \iota \hookrightarrow_{\mathsf{s}} (N, \vec{n} \cdot f(n)) \mathrel{-\!\!*} \Delta \vDash_{\mathcal{E}} K[n] \precsim e_2 : \tau}{\Delta \vDash_{\mathcal{E}} K[\,\mathsf{rand}(N)] \precsim e_2 : \tau}$$

REL-COUPLE-TAPES

$$\frac{\begin{array}{c} f \text{ bijection} \qquad e_1 \notin \mathit{Val} \qquad \iota \hookrightarrow (N, \vec{n}) \\ \iota' \hookrightarrow_{\mathsf{s}} (N, \vec{n}') \qquad \forall n \leq N.\, \iota \hookrightarrow (N, \vec{n} \cdot n) * \iota' \hookrightarrow_{\mathsf{s}} (N, \vec{n}' \cdot f(n)) \mathrel{-\!\!*} \Delta \vDash_{\mathcal{E}} e_1 \precsim e_2 : \tau \end{array}}{\Delta \vDash_{\mathcal{E}} e_1 \precsim e_2 : \tau}$$

Fig. 4. Coupling rules for the Clutch refinement judgment.

REL-NA-INV-OPEN

$$\frac{\mathcal{N} \in \mathcal{E} \qquad \boxed{P}^{\mathcal{N}} \qquad \triangleright P * \mathsf{closeNaInv}_{\mathcal{N}}(P) \mathrel{-\!\!*} \Delta \vDash_{\mathcal{E}\setminus\mathcal{N}} e_1 \precsim e_2 : \tau}{\Delta \vDash_{\mathcal{E}} e_1 \precsim e_2 : \tau}$$

REL-NA-INV-CLOSE

$$\frac{\triangleright P \qquad \mathsf{closeNaInv}_{\mathcal{N}}(P) \qquad \Delta \vDash_{\mathcal{E}} e_1 \precsim e_2 : \tau}{\Delta \vDash_{\mathcal{E}\setminus\mathcal{N}} e_1 \precsim e_2 : \tau}$$

REL-NA-INV-ALLOC

$$\frac{\triangleright P \qquad \boxed{P}^{\mathcal{N}} \mathrel{-\!\!*} \Delta \vDash_{\mathcal{E}} e_1 \precsim e_2 : \tau}{\Delta \vDash_{\mathcal{E}} e_1 \precsim e_2 : \tau}$$

Fig. 5. Non-atomic invariant access rules for the Clutch refinement judgment.

can mostly be ignored as we have done and will do throughout the paper. The later modality is essential for the soundness of the logical relation and taking guarded fixpoints $\mu x.\, P$ that require the recursive occurrence x to appear under the later modality, but our use is entirely standard. We refer to Jung et al. [2018] for more details on the later modality and how it is generally used in Iris.

5 MODEL OF CLUTCH

In this section we show how the connectives of Clutch are modeled through a shallow embedding in the *base logic* of the Iris separation logic [Jung et al. 2018]. First, we describe how we define a relational *coupling logic* (§5.1) that is used to establish couplings between programs. Next, we show how the coupling logic in combination with a binary interpretation of types is used to define the refinement logic (§5.2). Finally, we summarize how the final soundness theorem is proven (§5.3).

The general structure and skeleton of our model mimics the construction of several *non*-probabilistic logical relations found in prior work [Frumin et al. 2021b; Krebbers et al. 2017b; Turon et al. 2013a,b]. A key contribution and benefit of Clutch is that that same structure can be adapted to handle *probabilistic* refinements through the right choice of intermediate definitions and abstractions, as we will highlight throughout this section. While some aspects of the model are Iris-specific, the key ideas are general and should apply to other frameworks as well.

5.1 Coupling Logic

We recall that probabilistic couplings are used to prove relations between distributions by constructing a joint distribution that relates two distributions in a particularly desirable way:

Definition 5 (Coupling). *Let $\mu_1 \in \mathcal{D}(A)$, $\mu_2 \in \mathcal{D}(B)$. A sub-distribution $\mu \in \mathcal{D}(A \times B)$ is a* coupling *of μ_1 and μ_2 if*

(1) $\forall a.\ \sum_{b \in B} \mu(a, b) = \mu_1(a)$
(2) $\forall b.\ \sum_{a \in A} \mu(a, b) = \mu_2(b)$

Given a relation $R \subseteq A \times B$ we say μ is an R-coupling if furthermore $\mathrm{supp}(\mu) \subseteq R$. *We write $\mu_1 \sim \mu_2 : R$ if there exists an R-coupling of μ_1 and μ_2.*

Couplings can be constructed and composed along the monadic structure of sub-distributions.

Lemma 6 (Composition of couplings). *Let $R \subseteq A \times B$, $S \subseteq A' \times B'$, $\mu_1 \in \mathcal{D}(A)$, $\mu_2 \in \mathcal{D}(B)$, $f_1 : A \to \mathcal{D}(A')$, and $f_2 : B \to \mathcal{D}(B')$.*

(1) *If $(a, b) \in R$ then* $\mathrm{ret}(a) \sim \mathrm{ret}(b) : R$.
(2) *If $\mu_1 \sim \mu_2 : R$ and for all $(a, b) \in R$ it is the case that $f_1(a) \sim f_2(b) : S$ then $\mu_1 \gg\!= f_1 \sim \mu_2 \gg\!= f_2 : S$*

Once a coupling has been established, we can often extract a concrete relation from it between the probability distributions. In particular, for $(=)$-couplings, we have the following result.

Lemma 7. *If $\mu_1 \sim \mu_2 : (=)$ then $\mu_1 = \mu_2$.*

The Clutch coupling logic can be seen as a higher-order separation logic analogue of Barthe et al. [2015]'s pRHL logic. However, unlike pRHL, which uses the four-part Hoare *quadruples* that we saw in §1 to do relational reasoning, the coupling logic instead follows CaReSL [Turon et al. 2013a] and encodes one of the programs as a separation logic *ghost resource*. In particular, the coupling logic consists of two components: (1) a *unary* weakest precondition theory $\mathsf{wp}\ e\ \{\Phi\}$; and (2) a *specification resource* $\mathsf{spec}(e')$ with *specification context* specCtx. We think of the program e in the weakest precondition predicate as representing the program that occurs on the left side of a quadruple, while the specification program e' represents the right side program. The specification context assertion specCtx will be used to connect the weakest precondition to the specification resource. Ultimately, by showing

$$\mathsf{specCtx} * \mathsf{spec}(e') \vdash \mathsf{wp}\ e\ \{v.\exists v'.\ \mathsf{spec}(v') * \varphi(v, v')\}$$

in the logic, we will have established a φ-coupling of the executions of the programs e and e'.

The weakest precondition. The weakest precondition connective $\mathsf{wp}\ e\ \{v.\Phi\}$ is a new probabilistic weakest precondition that we formally define below. In isolation it simply means that the execution of e is safe (*i.e.*, the probability of crashing is zero), and for every possible return value v of e, the postcondition $\Phi(v)$ holds. Note however, that it encodes *partial* correctness, as it does not imply that the probability of termination is necessarily one, meaning the program may diverge.

In most Iris-style program logics, the weakest precondition $\mathsf{wp}\ e\ \{\Phi\}$ is a predicate stating that either the program e is a value satisfying Φ or it is reducible such that for any other term e' that it reduces to, then $\mathsf{wp}\ e'\ \{\Phi\}$ must hold as well. This guarantees safety of the full execution of the program e. The weakest precondition that we define in this section has—in isolation—the same intuition but it is fundamentally different. It is still a *unary* predicate, but in order to do relational reasoning, the weakest precondition pairs up the probability distribution of individual program steps of the left-hand side with the probability distribution of individual steps of *some* other program in such a way that there exists a *probabilistic coupling* among them. Through the specCtx we will guarantee that this "other" program is tied to the program tracked by the $\mathsf{spec}(e')$ resource. The weakest precondition itself satisfies all the usual structural rules such as wp-wand and wp-bind found in Figure 6 as well as language-level primitive rules such as wp-load, but in combination with the specCtx and $\mathsf{spec}(e')$ resources, the coupling logic satisfies rules like wp-couple-rands and wp-couple-tape-l. Notice the resemblance between wp-couple-rands and prhl-couple from §1.

The weakest precondition connective is given by a guarded fixpoint of the equation below—the fixpoint exists because the recursive occurrence appears under the later modality.[3]

$$\begin{aligned}\mathsf{wp}\ e_1\ \{\Phi\} \triangleq\ & (e_1 \in \mathit{Val} \wedge \Phi(e_1)) \vee \\ & (e_1 \notin \mathit{Val} \wedge \forall \sigma_1, \rho_1'.\ S(\sigma_1) * G(\rho_1') \mathrel{-\!\!*} \\ & \quad \mathsf{execCoupl}((e_1, \sigma_1), \rho_1')(\lambda(e_2, \sigma_2), \rho_2'.\ \triangleright S(\sigma_2) * G(\rho_2') * \mathsf{wp}_{\mathcal{E}}\ e_2\ \{\Phi\}))\end{aligned}$$

The base case says that if the expression e_1 is a value then the postcondition $\Phi(e_1)$ must hold. On the other hand, if e_1 is *not* a value, we get to assume two propositions $S(\sigma_1)$ and $G(\rho_1')$ for any $\sigma_1 \in \mathit{State}, {\rho_1}' \in \mathit{Cfg}$, and then we must prove $\mathsf{execCoupl}((e_1, \sigma_1), \rho_1')(\ldots)$. The $S : \mathit{State} \to \mathit{iProp}$ predicate is a *state interpretation* that interprets the state (the heap and the tapes) of the language as resources in Clutch and gives meaning to the $\ell \mapsto v$ and $\iota \hookrightarrow (N, \vec{n})$ connectives. The $G : \mathit{Cfg} \to$

[3] We omit from the definition occurrences of the Iris *fancy update modality* needed for resource updates and necessary book-keeping related to Iris invariants—these matters are essential but our use is entirely standard. For the Iris expert we refer to the appendix [Gregersen et al. 2023b] for the full definition.

WP-WAND

$$\frac{\forall v.\, \Phi(v) \mathrel{-\!\!*} \Psi(v) \qquad \mathsf{wp}\ e\ \{\Phi\}}{\mathsf{wp}\ e\ \{\Psi\}}$$

WP-BIND

$$\frac{\mathsf{wp}\ e\ \left\{v.\mathsf{wp}\ K[v]\ \{\Phi\}\right\}}{\mathsf{wp}\ K[e]\ \{\Phi\}}$$

WP-LOAD

$$\frac{\ell \mapsto v \qquad \ell \mapsto v \mathrel{-\!\!*} \Phi(v)}{\mathsf{wp}\ !\,\ell\ \{\Phi\}}$$

WP-COUPLE-RANDS

$$\frac{f \text{ bijection} \qquad \mathsf{specCtx} \qquad \mathsf{spec}(\mathsf{rand}(N)) \qquad \forall n \le N.\, \mathsf{spec}(f(n)) \mathrel{-\!\!*} \Phi(n)}{\mathsf{wp}\ \mathsf{rand}(N)\ \{\Phi\}}$$

WP-COUPLE-TAPE-L

$$\frac{\begin{array}{c} f \text{ bijection} \qquad e \notin \mathit{Val} \qquad \mathsf{specCtx} \\ \mathsf{spec}(\mathsf{rand}(N)) \qquad \iota \hookrightarrow (N, \vec{n}) \qquad \forall n \le N.\ (\mathsf{spec}(f(n)) * \iota \hookrightarrow (N, \vec{n} \cdot n)) \mathrel{-\!\!*} \mathsf{wp}\ e\ \{\Phi\} \end{array}}{\mathsf{wp}\ e\ \{\Phi\}}$$

Fig. 6. Selected structural rules of the weakest preconditon.

iProp predicate is a *specification interpretation* that allows us to interpret and track the "other" program that we are constructing a coupling with—we return to its instantiation momentarily.

The key technical novelty and the essence of the weakest precondition is the *coupling modality*: Intuitively, the proposition $\mathsf{execCoupl}(\rho_1, \rho_1')(\lambda \rho_2, \rho_2'.\, P)$ says that there exists a series of (composable) couplings starting from configurations ρ_1 and ρ_1' that ends up in some configurations ρ_2 and ρ_2' such that the proposition P holds. With this intuition in mind, the last clause of the weakest precondition says that the execution of (e_1, σ_1) can be coupled with the execution of ρ_1' such that the state and specification interpretations still hold for the end configurations, and the weakest precondition holds recursively for the continuation e_2.

Coupling modality. The coupling modality is an inductively defined proposition in Clutch, formally defined as a least fixpoint of an equation with six different disjuncts found in the appendix [Gregersen et al. 2023b]. The modality supports both synchronous and asynchronous couplings on both sides while ensuring that the left program takes at least one step. As it is inductively defined, we can chain together multiple couplings but it always ends in base cases that couple a single step of the left-hand side program—this aligns with the usual intuition that each unfolding of the recursively defined weakest precondition corresponds to one physical program step.

For instance, we can couple two physical program steps through the following constructor:

$$\frac{\mathsf{red}(\rho_1) \qquad \mathsf{step}(\rho_1) \sim \mathsf{step}(\rho_1') : R \qquad \forall \rho_2, {\rho_2}'.\, R(\rho_2, {\rho_2}') \mathrel{-\!\!*} Z(\rho_2, {\rho_2}')}{\mathsf{execCoupl}(\rho_1, \rho_1')(Z)}$$

Intuitively, this says that to show $\mathsf{execCoupl}(\rho_1, {\rho_1}')(Z)$ we (1) have to show that the configuration ρ_1 is *red*ucible which means that the program *can* take a step (this is to guarantee safety of the left-hand side program), (2) pick a relation R and show that there exists an R-coupling of the two program steps, and (3) for all configurations $\rho_2, {\rho_2}'$ in the support of the coupling, the logical predicate $Z(\rho_2, {\rho_2}')$ holds. This rule is used to justify the classical coupling rule WP-COUPLE-RANDS that (synchronously) couples two program samplings.

The coupling modality also allows to construct a coupling between a program step and a trivial (Dirac) distribution; this is used to validate proof rules that symbolically execute just one of the two sides. Indeed, the rule below allows us to progress the right-hand side independently from the left-hand side, but notice the occurrence of the coupling modality in the premise—this allows us to chain multiple couplings together in a single coupling modality.

$$\frac{\mathsf{ret}(\rho_1) \sim \mathsf{step}(\rho_1') : R \qquad \forall {\rho_2}'.\, R(\rho_1, {\rho_2}') \twoheadrightarrow \mathsf{execCoupl}(\rho_1, {\rho_2}')(Z)}{\mathsf{execCoupl}(\rho_1, {\rho_1}')(Z)}$$

To support *asynchronous* couplings, we introduce a *state step* reduction relation $\rightarrow_\iota \subseteq State \times [0,1] \times State$ that uniformly at random samples a natural number n to the end of the tape ι:

$$\sigma \rightarrow_\iota^{1/(N+1)} \sigma[\iota \rightarrow (N, \vec{n} \cdot n)] \qquad\qquad \text{if } \sigma(\iota) = (N, \vec{n}) \text{ and } n \leq N$$

Let $\mathsf{step}_\iota(\sigma)$ denote the induced distribution of a single state step reduction of σ. The coupling modality allows us to introduce couplings between $\mathsf{step}_\iota(\sigma)$ and a sampling step:

$$\frac{\mathsf{step}_\iota(\sigma_1) \sim \mathsf{step}({\rho_1}') : R \qquad \forall \sigma_2, {\rho_2}'.\, R(\sigma_2, {\rho_2}') \twoheadrightarrow \mathsf{execCoupl}((e_1, \sigma_2), {\rho_2}')(Z)}{\mathsf{execCoupl}((e_1, \sigma_1), {\rho_1}')(Z)}$$

Note that here the left-hand side program does not take a physical step, thus the coupling modality appears in the premise as well. This particular rule is key to the soundness of the asynchronous coupling rule WP-COUPLE-TAPE-L that couples a sampling to a tape on the left with a program sampling on the right. We use similar constructors of execCoupl to prove, *e.g.* REL-COUPLE-TAPE-R. The crux is, however, that the extra state steps that we inject in the coupling modality to prove the asynchronous coupling rules *do not matter* (!) in the sense that they can be entirely erased as part of the coupling logic's adequacy theorem (Theorem 11).

A specification resource and context with run ahead. We will encode a relational specification into a unary specification by proving a unary weakest precondition about e (the *implementation*), in which e' (the *specification*) is tracked using a ghost resource $\mathsf{spec}(e')$ that can be updated to reflect execution steps. The ghost specification connective $\mathsf{spec}(e')$, together with the specCtx proposition, satisfies a number of symbolic execution rules following the operational semantics.

The specCtx proposition is an Iris invariant and its purpose is twofold: (1) it gives meaning to the ghost specification resource $\mathsf{spec}(e)$ and the heap and tape assertions, $\ell \mapsto_{\mathsf{s}} v$ and $\iota \hookrightarrow_{\mathsf{s}} (N, \vec{n})$, and (2) it connects the $\mathsf{spec}(e)$ resource to the program e' that we are constructing a coupling with in the weakest precondition. We keep track of e' through the specification interpretation G. When constructing a final closed proof we will want e to be equal to e', however, during proofs they are not always going to be the same—we will allow e to *run ahead* of e'. As a consequence, it will be possible to reason *independently* about the right-hand side without consideration of the left-hand side as exemplified by the rules below[4], that allow us to progress the specification program but without considering the weakest precondition or the left-hand side program.

SPEC-PURE

$$\frac{\mathsf{specCtx} \qquad \mathsf{spec}(K[e]) \qquad e \overset{\mathsf{pure}}{\rightsquigarrow} e'}{\mathsf{spec}(K[e'])}$$

SPEC-STORE

$$\frac{\mathsf{specCtx} \qquad \mathsf{spec}(K[\ell \leftarrow w]) \qquad \ell \mapsto_{\mathsf{s}} v}{\mathsf{spec}(K[()]) * \ell \mapsto_{\mathsf{s}} w}$$

Similarly looking rules exists for all the deterministic right-hand side reductions.

To define specCtx we will use two instances of the *authoritative resource algebra* [Jung et al. 2015] from the Iris ghost theory. It suffices to know that an instance F gives us two resources $F_\bullet(a)$ and $F_\circ(a)$ satisfying $F_\bullet(a) * F_\circ(b) \vdash a = b$ and that $F_\bullet(a) * F_\circ(b)$ can be updated to $F_\bullet(a') * F_\circ(a')$. To connect the two parts we will keep $\mathsf{specInterp}_\bullet(\rho)$ in the specification interpretation G (that

[4] Technically, the consequence of the rules is under a fancy update modality that we omit for the sake of presentation.

"lives" in the weakest precondition), and the corresponding $\mathsf{specInterp}_\circ(\rho)$ in specCtx:

$$\begin{aligned}
G(\rho) &\triangleq \mathsf{specInterp}_\bullet(\rho) \\
\mathsf{specInv} &\triangleq \exists \rho, e, \sigma, n.\ \mathsf{specInterp}_\circ(\rho) * \mathsf{spec}_\bullet(e) * \mathsf{heaps}(\sigma) * \ \mathsf{execConf}_n(\rho)(e, \sigma) = 1 \\
\mathsf{specCtx} &\triangleq \boxed{\mathsf{specInv}}^{\mathcal{N}.\mathsf{spec}}
\end{aligned}$$

This ensures that the configuration ρ tracked in the weakest precondition is the same as the configuration ρ tracked in specCtx. On top of this, specCtx contains resources $\mathsf{spec}_\bullet(e)$ and $\mathsf{heaps}(\sigma)$ while guaranteeing that the configuration (e, σ) can be reached in n deterministic program steps from ρ. The $\mathsf{heaps}(\sigma)$ resource gives meaning—using standard Iris ghost theory—to the heap and tape assertions, $\ell \mapsto_{\mathsf{s}} v$ and $\iota \hookrightarrow_{\mathsf{s}} (N, \vec{n})$, just like the state interpretation in the weakest precondition. $\mathsf{execConf}_n : \mathit{Cfg} \to \mathcal{D}(\mathit{Cfg})$ denotes the distribution of n-step partial execution. By letting $\mathsf{spec}(e) \triangleq \mathsf{spec}_\circ(e)$ this construction permits the right-hand side program to progress (with deterministic reduction steps) without consideration of the left-hand side as exemplified by SPEC-PURE and SPEC-STORE. However, when applying coupling rules that actually need to relate the two sides, the proof first "catches up" with $\mathsf{spec}(e)$ using the execCoupl rule that progresses the right-hand side independently, before constructing the coupling of interest.

5.2 Refinement Logic

Contextual refinement is a typed relation and hence logical refinement must be typed as well. To define the refinement logic, we first define a binary value interpretation $[\![\tau]\!]_\Delta$ that characterizes the set of pairs of closed values (v_1, v_2) of type τ such that v_1 contextually refines v_2. The definition follows the usual structure of ("logical") logical relations, see, *e.g.*, Frumin et al. [2021b]; Timany et al. [2022], by structural recursion on τ and uses corresponding logical connectives. Functions are interpreted via (separating) implication, universal types are interpreted through universal quantification, *etc.*, as found in the appendix [Gregersen et al. 2023b]. The only novelty is the interpretation of the new type of tapes shown below:

$$[\![\mathsf{tape}]\!]_\Delta(v_1, v_2) \triangleq \exists \iota_1, \iota_2, N.\ (v_1 = \iota_1) * (v_2 = \iota_2) * \boxed{\iota_1 \hookrightarrow (N, \epsilon) * \iota_2 \hookrightarrow_{\mathsf{s}} (N, \epsilon)}^{\mathcal{N}.\iota_1.\iota_2}$$

The interpretation requires that the values are tape labels, *i.e.*, references to tapes, and that they are always empty as captured by the invariant. Intuitively, this guarantees through coupling rules and the symbolic execution rules from Figure 3 that we always can couple samplings on these tapes as needed in the compatibility lemma for T-RAND as discussed in §5.3. Point-wise equality of the two tapes would also have been sufficient for the compatibility lemma but by requiring them to be empty we can prove general equivalences such as $\iota : \mathsf{tape} \vdash \mathsf{rand}(N) \simeq_{\mathrm{ctx}} \mathsf{rand}(N, \iota) : \mathsf{nat}$.

The refinement judgment is defined using the coupling logic in combination with the binary value interpretation. Recall how the intuitive reading of the refinement judgment $\Delta \vDash_{\mathcal{E}} e_1 \precsim e_2 : \tau$ is that the expression e_1 refines the expression e_2 at type τ under the invariants in the mask $\mathcal{E}$ with interpretations of type variables in τ taken from Δ. Besides the coupling logic and the binary value interpretations, we will also make use of the resource $\mathsf{naTok}(\mathcal{E})$ that keeps track of the set of non-atomic invariants that are currently *closed.*

Putting everything together, the refinement judgment is formally defined as follows:

$$\begin{aligned}
\Delta \vDash_{\mathcal{E}} e_1 \precsim e_2 : \tau \triangleq \forall K.\ &\mathsf{specCtx} \twoheadrightarrow \mathsf{spec}(K[e_2]) \twoheadrightarrow \mathsf{naTok}(\mathcal{E}) \twoheadrightarrow \\
&\mathsf{wp}\ e_1 \left\{ v_1. \exists v_2.\ \mathsf{spec}(K[v_2]) * \mathsf{naTok}(\top) * [\![\tau]\!]_\Delta(v_1, v_2) \right\}
\end{aligned}$$

The definition assumes that the right-hand side program is executing e_2 and that the invariants in $\mathcal{E}$ are closed, and it concludes that the two executions can be aligned so that if e_1 reduces to some value v_1 then there exists a corresponding execution of e_2 to a value v_2 and all invariants have been closed. Moreover, the values v_1 and v_2 are related via the binary value interpretation $[\![\tau]\!]_\Delta(v_1, v_2)$. By quantifying over K, we close the definition under evaluation contexts on the right-hand side. For the left-hand side this is not needed as the weakest precondition already satisfies wp-bind.

5.3 Soundness

The soundness of the refinement judgment hinges on the soundness of the coupling logic. The goal of the coupling logic is to show a *coupling* of the execution of the two programs, but to establish a coupling of two distributions they must have the same mass. Intuitively, due to the approximative nature of step-indexed logics like Clutch, we need to show—at every logical step-index—that a coupling exists, even when the left-hand side program has not yet terminated. This means we might not have enough mass on the left-hand side to cover all of the mass on the right-hand side. For this reason we introduce a new notion of *left-partial coupling*.

Definition 8 (Left-Partial Coupling). *Let $\mu_1 \in \mathcal{D}(A), \mu_2 \in \mathcal{D}(B)$. A sub-distribution $\mu \in \mathcal{D}(A \times B)$ is a* left-partial coupling *of μ_1 and μ_2 if*

(1) $\forall a.\ \sum_{b \in B} \mu(a, b) = \mu_1(a)$
(2) $\forall b.\ \sum_{a \in A} \mu(a, b) \leq \mu_2(b)$

Given a relation $R \subseteq A \times B$ we say μ is an R-left-partial-coupling if furthermore $\mathrm{supp}(\mu) \subseteq R$. We write $\mu_1 \lesssim \mu_2 : R$ if there exists an R-left-partial-coupling of μ_1 and μ_2.

This means that, for any $\mu \in \mathcal{D}(B)$ and any $R \subseteq A \times B$, the zero distribution $\mathbf{0}$ trivially satisfies $\mathbf{0} \lesssim \mu : R$. This reflects the asymmetry of both contextual refinement and our weakest precondition—it allows us to show that a diverging program refines any other program of appropriate type.

Left-partial couplings can also be constructed and composed along the monadic structure of the sub-distribution monad and are implied by regular couplings:

Lemma 9. *If $\mu_1 \sim \mu_2 : R$ then $\mu_1 \lesssim \mu_2 : R$.*

Additionally, proving a (=)-left-partial-coupling coincides with the point-wise inequality of distributions that will allow us to reason about contextual refinement.

Lemma 10. *If $\mu_1 \lesssim \mu_2 : (=)$ then $\forall a.\ \mu_1(a) \leq \mu_2(a)$.*

The *adequacy* theorem of the coupling logic is stated using left-partial couplings.

Theorem 11 (Adequacy). *Let $\varphi : Val \times Val \to Prop$ be a predicate on values in the meta-logic. If*

$$\mathsf{specCtx} * \mathsf{spec}(e') \vdash \mathsf{wp}\ e\ \{v.\exists v'.\ \mathsf{spec}(v') * \varphi(v, v')\}$$

is provable in Clutch then $\forall n.\ \mathrm{exec}_n(e, \sigma) \lesssim \mathrm{exec}(e', \sigma') : \varphi$.

As a simple corollary, contextual refinement follows from continuity of exec_n.

The proof of the adequacy theorem goes by induction in both n and the execCoupl fixpoint, followed by a case distinction on the big disjunction in the definition of execCoupl. Most cases are simple coupling compositions along the monadic structure except the cases where we introduce state step couplings that rely on *erasure* in the following sense:

Lemma 12 (Erasure). *If $\sigma_1(\iota) \in \mathrm{dom}(\sigma_1)$ then*

$$\mathrm{exec}_n(e_1, \sigma_1) \sim (\mathrm{step}_\iota(\sigma_1) \gg\!= \lambda\sigma_2.\ \mathrm{exec}_n(e_1, \sigma_2)) : (=)$$

Intuitively, this lemma tells us that we can prepend any program execution with a state step reduction and it will not have an effect on the final result. The idea behind the proof is that if we append a sampled value n to the end of a tape, and if we eventually consume n, then we obtain the same distribution as if we never appended n in the first place. This is a property that one should *not* take for granted: the operational semantics has been carefully defined such that reading from an empty tape reduces to a value as well, and none of the other program operations can alter or observe the contents of the tape. This ensures that presampled values are untouched until consumed and that the proof and the execution is independent.

To show the soundness theorem of the refinement logic, we extend the interpretation of types to typing contexts—$[\![\Gamma]\!]_\Delta(\vec{v}, \vec{w})$ iff for every $x_i : \sigma_i$ in Γ then $[\![\sigma_i]\!]_\Delta(v_i, w_i)$ holds—and the refinement judgment to open terms by closing substitutions as usual:

$$\Delta \mid \Gamma \vDash e_1 \precsim e_2 : \tau \triangleq \forall \vec{v}, \vec{w}.\ [\![\Gamma]\!]_\Delta(\vec{v}, \vec{w}) \twoheadrightarrow \Delta \vDash e_1[\vec{v}/\Gamma] \precsim e_2[\vec{w}/\Gamma] : \tau$$

where $e_1[\vec{v}/\Gamma]$ denotes simultaneous substitution of every x_i from Γ in e_1 by the value v_i.

We then show, using the structural and symbolic execution rules of the refinement judgment, that the typing rules are *compatible* with the relational interpretation: for every typing rule, if we have a pair of related terms for every premise, then we also have a pair of related terms for the conclusion. See for instance the compatibility rule for T-RAND below in the case $\tau = \mathsf{tape}$ that follows using REL-BIND and REL-COUPLE-TAPES.

RAND-COMPAT

$$\frac{\Delta \mid \Gamma \vDash e_1 \precsim e_1' : \mathsf{nat} \qquad \Delta \mid \Gamma \vDash e_2 \precsim e_2' : \mathsf{tape}}{\Delta \mid \Gamma \vDash \mathsf{rand}(e_1, e_2) \precsim \mathsf{rand}(e_1', e_2') : \mathsf{nat}}$$

As a consequence of the compatibility rules, we obtain the fundamental theorem of logical relations.

Theorem 13 (Fundamental theorem). *Let $\Xi \mid \Gamma \vdash e : \tau$ be a well-typed term, and let Δ assign a relational interpretation to every type variable in Ξ. Then $\Delta \mid \Gamma \vDash e \precsim e : \tau$.*

The compatibility rules, moreover, yield that the refinement judgment is a congruence, and together with Theorem 11 we can then recover contextual refinement:

Theorem 14 (Soundness). *Let Ξ be a type variable context, and assume that, for all Δ assigning a relational interpretation to all type variables in Ξ, we can derive $\Delta \mid \Gamma \vDash e_1 \precsim e_2 : \tau$. Then $\Xi \mid \Gamma \vdash e_1 \precsim_{ctx} e_2 : \tau$*

6 CASE STUDIES

In the coming sections, we give an overview of some of the example equivalences we have proven with Clutch. Further details are found in the appendix [Gregersen et al. 2023b] and our Coq development. In particular, in the appendix [Gregersen et al. 2023b] we discuss an example by Sangiorgi and Vignudelli [2016], which previous probabilistic logical relations without asynchronous couplings could not prove [Bizjak 2016, Sec. 1.5].

6.1 Lazy/Eager Coin

In this section we give a more detailed proof of the lazy-eager coin example from §1. We will go through the proof step by step but omit the use of REL-PURE-L and REL-PURE-R which should be interleaved with the application of most of the mentioned proof rules.

Recall the definitions of *lazy* and *eager* from §1. The goal is to show $\vdash \mathit{lazy} \simeq_{\mathrm{ctx}} \mathit{eager} : \mathsf{unit} \to \mathsf{bool}$ by first showing $\mathit{lazy} \precsim_{\mathrm{ctx}} \mathit{eager} : \mathsf{unit} \to \mathsf{bool}$ and then $\mathit{eager} \precsim_{\mathrm{ctx}} \mathit{lazy} : \mathsf{unit} \to \mathsf{bool}$.

To show $\mathit{lazy} \precsim_{\mathrm{ctx}} \mathit{eager} : \mathsf{unit} \to \mathsf{bool}$, we first define an intermediate labeled version lazy' of *lazy* (found in §2). By transitivity of contextual refinement and Theorem 1 it is sufficient to show $\vDash \mathit{lazy} \precsim \mathit{lazy}' : \mathsf{unit} \to \mathsf{bool}$ and $\vDash \mathit{lazy}' \precsim \mathit{eager} : \mathsf{unit} \to \mathsf{bool}$.

The first refinement $\vDash \mathit{lazy} \precsim \mathit{lazy}' : \mathsf{unit} \to \mathsf{bool}$ is mostly straightforward. By applying REL-ALLOC-L followed by REL-ALLOC-TAPE-R and REL-ALLOC-R we are left with the goal of proving that the two thunks are related, given $\iota \hookrightarrow_{\mathsf{s}} (1, \epsilon)$, $\ell \mapsto \mathsf{None}$ and $\ell' \mapsto_{\mathsf{s}} \mathsf{None}$ for some fresh label ι and fresh locations on the heap ℓ and ℓ'. Using rel-na-inv-alloc we allocate the invariant

$$\begin{aligned}\iota \hookrightarrow_{\mathsf{s}} (1, \epsilon) * ((\ell \mapsto \mathsf{None} \; * \ell' \mapsto_{\mathsf{s}} \mathsf{None}) \\ \vee (\exists b.\, \ell \mapsto \mathsf{Some}(b) * \ell' \mapsto_{\mathsf{s}} \mathsf{Some}(b)))\end{aligned}$$

with some name $\mathcal{N}$ that expresses how the ι tape is always empty and that *either* both ℓ and ℓ' contain None *or* both contain Some(b) for some b. We continue by rel-rec after which we open the invariant and do a case distinction on the disjunction in the invariant. If ℓ and ℓ' are empty, this is the first time we invoke the function. We continue using rel-load-l and rel-load-r after which we are left with the goal

$$\vDash_{\top\setminus\mathcal{N}} \begin{array}{l}\mathsf{let}\ b = \mathsf{flip}()\ \mathsf{in}\\ r \leftarrow \mathsf{Some}(b); b\end{array} \precsim \begin{array}{l}\mathsf{let}\ b = \mathsf{flip}(\iota)\ \mathsf{in}\\ r \leftarrow \mathsf{Some}(b); b\end{array} : \mathsf{unit} \to \mathsf{bool}$$

We continue using rel-rand-erase-r to couple the two flips, we follow by rel-store-l and rel-store-r to store the fresh bit on the heaps, we close the invariant (now showing the right disjunct as the locations have been updated) using rel-na-inv-close, and we finish the case using rel-return as the program returns the same Boolean b on both sides.

If ℓ and ℓ' were *not* empty, this is not the first time the function is invoked and we straightforwardly load the same Boolean on both sides using rel-load-l and rel-load-r and finish the proof using rel-na-inv-close and rel-return.

For the second refinement $\vDash$ *lazy*$'$ $\precsim$ *eager* : unit $\to$ bool we start by allocating the tape on the left using rel-alloc-tape-l which gives us ownership of a fresh tape $\iota \hookrightarrow (1, \epsilon)$. We now couple the ι tape with the unlabeled flip() on the right using rel-couple-tape-l. This gives us that for some b then $\iota \hookrightarrow (1, b)$ and the flip() on the right returned b as well. We continue by allocating the reference on the left using rel-alloc-l which gives us some location ℓ and $\ell \mapsto \mathsf{None}$. Now, we allocate the invariant

$$(\iota \hookrightarrow (1, b) * \ell \mapsto \mathsf{None}) \vee \ell \mapsto \mathsf{Some}(b)$$

which expresses that *either* the location ℓ is empty but b is on the ι tape, *or* b has been stored at ℓ. We are now left with proving that the two thunks are related under this invariant. We continue using rel-rec after which we open the invariant using rel-na-inv-open, do a case distinction on the disjunction, and continue using rel-load-l. If the location ℓ is empty, we have to show

$$\vDash_{\top\setminus\mathcal{N}} \begin{array}{l}\mathsf{let}\ b = \mathsf{flip}(\iota)\ \mathsf{in}\\ r \leftarrow \mathsf{Some}(b); b\end{array} \precsim b : \mathsf{unit} \to \mathsf{bool}$$

But as we own $\iota \hookrightarrow (1, b)$ we continue using rel-rand-tape-l, rel-store-l, rel-na-inv-close (now establishing the right disjunct as ℓ has been updated), and rel-return as the return value b is the same on both sides. If the location ℓ was *not* empty, we know $\ell \mapsto \mathsf{Some}(b)$ which means rel-load-l reads b from ℓ and we finish the proof using rel-na-inv-close and rel-return.

The proof of *eager* $\precsim_{\text{ctx}}$ *lazy* : unit $\to$ bool is analogous and we have shown the contextual equivalence of the programs *eager* and *lazy*.

6.2 ElGamal Public Key Encryption

An encryption scheme is seen as secure if no probabilistic polynomial-time (PPT) adversary $\mathcal{A}$ can break it with non-negligible probability. A common pattern in cryptographic security proofs are security reductions. To perform a reduction, one assumes that such an adversary $\mathcal{A}$ exists, and constructs another PPT adversary $\mathcal{B}$ that, using $\mathcal{A}$, solves a computational problem P that is believed to be hard. By contradiction, this means the construction is secure under the assumption

```
keygen ≜ λ _. let sk = rand(n) in
              let pk = g^sk in
              (sk, pk)
  dec ≜ λ sk (B, X). X · B^(−sk)

enc ≜ λ pk msg. let b = rand(n) in
                let B = g^b in
                let X = msg · pk^b in
                (B, X)
```

Fig. 7. The ElGamal public key scheme.

```
PK_real ≜
let (sk, pk) = keygen() in
let count = ref 0 in
let query = λ msg.
  if !count ≠ 0 then
    None
  else
    count ← 1;


    let (B, X) = enc pk msg in
    Some (B, X)
 in (pk, query)
```

```
PK_rand ≜
let (sk, pk) = keygen() in
let count = ref 0 in
let query = λ msg.
  if !count ≠ 0 then
    None
  else
    count ← 1;
    let b = rand(n) in
    let x = rand(n) in
    let (B, X) = (g^b, g^x) in
    Some (B, X)
 in (pk, query)
```

(a) The security games.

```
C[−] ≜
let (pk, B, C) = − in
let count = ref 0 in
let query = λ msg.
  if !count ≠ 0 then
    None
  else
    count ← 1;


    let X = msg · C in
    Some (B, X)
 in (pk, query)
```

(b) The DH reduction context.

Fig. 8. Public key security.

that the problem P is hard. A crucial proof step is showing that $\mathcal{B}$ together with P corresponds to the original construction which can be thought of as the "soundness" of the security reduction. In this section, we use Clutch to show the soundness of a security reduction of the ElGamal public key encryption scheme [Elgamal 1985] to the decisional Diffie-Hellman (DDH) computational assumption.

The ElGamal construction is a public key encryption scheme consisting a tuple of algorithms $(keygen, enc, dec)$ whose implementation in $\mathbf{F}^{\text{rand}}_{\mu,\text{ref}}$ is shown in Figure 7. The implementation is parameterized by a group G which serves to represent messages, ciphertexts, and keys. We write $G = (1, \cdot, -^{-1})$ for a finite cyclic group of order $|G|$, generated by g, and let $n = |G| - 1$. Intuitively, to show that ElGamal encryption is secure it suffices to show that, given the DDH assumption holds for the group G, an adversary $\mathcal{A}$ cannot distinguish an encrypted message from a random ciphertext (see, *e.g.*, [Rosulek 2020, §15.3]). The DDH assumption for a group G says that the two games DH_{real} and DH_{rand} in Figure 9 are PPT-indistinguishable which intuitively means that the value g^{ab} looks random, even to someone who has seen g^a and g^b.

The intuitive notion of encryption scheme security can be made precise[5] as the indistinguishability of two security games, *i.e.*, stylized interactions, PK_{real} and PK_{rand} shown in Figure 8, by a PPT[6] adversary. Here we interpret the notion of an "adversary" as a program context. Both security

[5] Several formulations exist in the literature; we take inspiration from the textbook presentation of Rosulek [2020].
[6] Polynomial-time with respect to the security parameter, *i.e.* the logarithm of the size of the group for ElGamal.

$$
\begin{aligned}
DH_{real} \triangleq\ & \mathsf{let}\ a = \mathsf{rand}(n)\ \mathsf{in} \\
& \mathsf{let}\ b = \mathsf{rand}(n)\ \mathsf{in} \\
& (g^a, g^b, g^{ab})
\end{aligned}
\qquad\qquad
\begin{aligned}
DH_{rand} \triangleq\ & \mathsf{let}\ a = \mathsf{rand}(n)\ \mathsf{in} \\
& \mathsf{let}\ b = \mathsf{rand}(n)\ \mathsf{in} \\
& \mathsf{let}\ c = \mathsf{rand}(n)\ \mathsf{in} \\
& (g^a, g^b, g^c)
\end{aligned}
$$

Fig. 9. The Decisional Diffie-Hellman game.

games are initialised by generating a secret/public-key pair (sk, pk), of which pk is returned to the adversary (the context). The adversary gets to examine the public key and an "encryption oracle" *query*, *i.e.*, a partial application of the encryption function specialized to a particular key. The difference between PK_{real} and PK_{rand} lies in the *query* function. While PK_{real} encrypts the message *msg* provided as input, PK_{rand} instead returns a randomly sampled ciphertext. Both games use a counter *count* to ensure that the *query* oracle can be called only once. One attempt at distinguishing the security games will thus correspond exactly to one attempt at distinguishing DH_{real} from DH_{rand}.

The idea now is to use Clutch as a step towards reducing indistinguishability of PK_{real} and PK_{rand} to the indistinguishability of DH_{real} and DH_{rand}. Specifically, we will exhibit a context C and show

$$\vdash PK_{real} \simeq_{\mathrm{ctx}} C[DH_{real}] : \tau_{PK} \tag{1}$$

$$\vdash PK_{rand} \simeq_{\mathrm{ctx}} C[DH_{rand}] : \tau_{PK} \tag{2}$$

Then we can complete the reduction on paper (outside of Clutch) as follows.[7] To prove that the DDH assumption implies public key security, we assume the contrapositive, *i.e.*, that there exists an adversarial context $\mathcal{A}$ that can distinguish PK_{real} from PK_{rand}. Using (1) and (2) we then get that $\mathcal{A}$ can distinguish $C[DH_{real}]$ from $C[DH_{rand}]$. But this means that $\mathcal{A}[C[-]]$ is a context that can distinguish the DDH games, and hence contradicts our assumption, if $\mathcal{A}[C[-]]$ is PPT. The context C for (1) and (2) is given by Figure 8b (note that the hole is in the first line). The proof that $\mathcal{A}[C[-]]$ is PPT if $\mathcal{A}[-]$ is PPT is outside of the scope of Clutch.

We will only focus on the first equation (1), since the proof of (2) is similar. The proof proceeds via an intermediate program, PK^{tape}_{real}, which differs from PK_{real} only in that the random sampling in *query* is labelled with the tape β. By transitivity, it suffices to show that $\vdash PK_{real} \simeq_{\mathrm{ctx}} PK^{tape}_{real} : \tau$ and $\vdash PK^{tape}_{real} \simeq_{\mathrm{ctx}} C[DH_{real}] : \tau$, as displayed in Figure 10. The first equivalence is trivial. The essential difference between PK^{tape}_{real} and $C[DH_{real}]$ is that the *query* function in PK^{tape}_{real} samples b lazily, whereas in $C[DH_{real}]$, the sampling of b occurs eagerly in the beginning. The proof now proceeds in a manner similar to the lazy-eager coin example; details can be found in the formalization.

Clutch is well-suited for proving the soundness of the reduction for two reasons. Firstly, any public key encryption scheme can only be secure if it employs randomized encryption [Goldwasser and Micali 1984]. Dealing with randomization is thus unavoidable. Secondly, reasoning about the encryption oracle involves moving the random sampling used in the encryption across a function boundary (the *query* oracle) as we saw. This part of the argument crucially relies on asynchronous couplings. Systems like EasyCrypt and CertiCrypt handle this part of the argument through special-purpose rules for swapping statements that allows moving the random sampling outside the function boundary. However, it crucially relies on the fact that these works consider first-order languages with global state and use syntactic criteria and assertions on memory disjointness.

[7] To mechanize the argument one would need to formalize a notion of PPT and a proof that the context is in fact PPT which is out of scope for the work at hand.

```
     PK_real          ≃ctx        PK_real^tape        ≃ctx       C[DH_real]
                             let β = tape(n) in               let (pk, B, C) =
let sk = rand(n) in          let sk = rand(n) in                let a = rand(n) in
let pk = g^sk in             let pk = g^sk in                   let b = rand(n) in
                                                                (g^a, g^b, g^ab) in
let count = ref 0 in         let count = ref 0 in             let count = ref 0 in
let query = λ msg.           let query = λ msg.               let query = λ msg.
  if !count ≠ 0 then           if !count ≠ 0 then               if !count ≠ 0 then
    None                         None                             None
  else                         else                             else
    count ← 1;                   count ← 1;                       count ← 1;
    let b = rand(n) in           let b = rand(n, β) in
    let B = g^b in               let B = g^b in
                                 let C = pk^b in
    let X = msg · pk^b in        let X = msg · C in               let X = msg · C in
    Some (B, X)                  Some (B, X)                      Some (B, X)
 in (pk, query)              in (pk, query)                   in (pk, query)
```

Fig. 10. The “real” direction of the security reduction.

Note moreover that our security formulation makes crucial use of the fact that $\mathbf{F}^{\text{rand}}_{\mu,\text{ref}}$ is higher-order, randomized, and supports local state to return the *query* closure as a first class value. This allows us to capture the textbook cryptographic notion of adversaries and of a (closed-box) “oracle” precisely using standard notions such as higher-order functions and contextual equivalence, without introducing special linguistic and logical categories of adversaries parameterized by a set of oracles.

6.3 Hash Functions

When analyzing data structures that use hash functions, one commonly models the hash function under the *uniform hash assumption* or the *random oracle model* [Bellare and Rogaway 1993]. That is, a hash function h from a set of keys K to values V behaves as if, for each key k, the hash $h(k)$ is randomly sampled from a uniform distribution over V, independently of all the other keys. Of course, hash functions are not known to satisfy this assumption perfectly, but it can nevertheless be a useful modeling assumption for analyzing programs that use hashes.

The function *eager_hash* in Figure 11 encodes such a model of hash functions in $\mathbf{F}^{\text{rand}}_{\mu,\text{ref}}$. (We explain the reason for the “eager” name later.) Given a non-negative integer n, executing *eager_hash n* returns a hash function with $K = \{0, \ldots, n\}$ and $V = \mathbb{B}$. To do so, it initializes a mutable map m and then calls *sample_all*, which samples a Boolean b with flip for each key k and stores the results in m. These Booleans serve as the hash values. On input k, the hash function returned by *eager_hash* looks up k in the map m and returns the result, with a default value of false if $k \notin K$.

However, this model of uniform hash functions can be inconvenient for proofs because all of the random hash values are sampled *eagerly* when the function is initialized. To overcome this, an important technique in pencil-and-paper proofs is to show that the hash values can be sampled *lazily* (see, *e.g.*, Mittelbach and Fischlin [2021]). That is, we only sample a key k's hash value when

```
eager_hash ≜
  λn. let m = init_map () in
      sample_all m (n + 1);
      (λk. match get m k with
             Some(b) ⇒ b
           | None    ⇒ false
           end)
```

```
lazy_hash ≜
  λn. let vm = init_map () in
      let tm = init_map () in
      alloc_tapes tm (n + 1);
      (λk. match get vm k with
             Some(b) ⇒ b
           | None    ⇒ match get tm k with
                          Some(ι) ⇒
                            let b = flip(ι) in
                            set vm b; b
                        | None ⇒ false
                        end
           end)
```

Fig. 11. Eager and lazy models of hash functions.

it is hashed for the first time. This lets us more conveniently couple that sampling step with some step in another program.

Motivated by applications to proofs in cryptography, Almeida et al. [2019] formalized in EasyCrypt a proof of equivalence between an eager and lazy random oracle. Although sufficient for their intended application, this proof was done in the context of a language that uses syntactic restrictions to model the hash function's private state. To the best of our knowledge, no such equivalence proof between lazy and eager sampling has previously been given for a language with higher-order state and general references.

As an application of Clutch, we prove such an equivalence in $\mathbf{F}^{\text{rand}}_{\mu,\text{ref}}$. The function *lazy_hash* shown in Figure 11 encodes the lazy sampling version of the random hash generator. For its internal state, the lazy hash uses two mutable maps: the tape map *tm* stores tapes to be used for random sampling, and the value map *vm* stores the previously sampled values for keys that have been hashed. After initializing these maps, it calls *alloc_tapes*, which allocates a tape for each key $k \in K$ and stores the associated tape in *tm*, but does not yet sample hashes for any keys. The hash function returned by *lazy_hash* determines the hash for a key k in two stages. It first looks up k in *vm* to see if k already has a previously sampled hash value, and if so, returns the found value. Otherwise, it looks up k in the tape map *tm*. If no tape is found, then k must not be in K, so the function returns false. If a tape ι is found, then the code samples a Boolean b from this tape with flip, stores b for the key k in *vm*, and then returns b.

We prove that the eager and lazy versions are contextually equivalent, that is, $\vdash$ *eager_hash* $n \simeq_{\text{ctx}}$ *lazy_hash* $n : \text{int} \to \text{bool}$. The core idea behind this contextual equivalence proof is to maintain an invariant between the internal state of the two hash functions. Let m be the internal map used by the eager hash and let *tm* and *vm* be the tape and value maps, respectively, for the lazy hash. Then, at a high level, the invariant maintains the following properties:

(1) $\text{dom}(m) = \text{dom}(tm) = \{0, \ldots, n\}$.
(2) For all $k \in \{0, \ldots, n\}$, if $m[k] = b$ then either
  (a) $vm[k] = b$, or
  (b) $vm[k] = \bot$ and $tm[k] = \iota$ for some tape label ι such that $\iota \hookrightarrow (1, b)$.

Case (a) and (b) of the second part of this invariant capture the two possible states each key k can be in. Either the hash of k has been looked up before (case a), and so the sampled value stored

```
init_hash_rng ≜
  λ_. let f = lazy_hash MAX in
      let c = ref 0 in
      (λ_. let n = ! c in
           let b = f n in
           c ← n + 1; b)
```

(a) Hashing random number generator.

```
init_bounded_rng ≜
  λ_. let c = ref 0 in
      (λ_. let n = ! c in
           let b = if n ≤ MAX then flip()
                   else false in
           c ← n + 1; b)
```

(b) Bounded random number generator.

Fig. 12. Random number generators.

in vm must match that of m, or it has not been looked up (case b) and the tape for the key must contain the same value as $m[k]$ for its next value.

To establish this invariant when the hashes are initialized, we asynchronously couple the eager hash function's flip for key k with a tape step for the tape ι associated with k in the lazy table. The invariant ensures that the values returned by the two hash functions will be the same when a key k is queried. The cases of the invariant correspond to the branches of the lazy function's match statements: if the key k is in K and has been queried before, the maps will return the same values found in m and vm. If it has not been queried before, then flip in the lazy version will sample the value on the tape for the key, which matches $m[k]$. Moreover, the update that writes this sampled value to vm preserves the invariant, switching from case (b) to case (a) for the queried key.

We have used this more convenient lazy encoding to verify examples that use hash functions. For instance, one scheme to implement random number generators is to use a cryptographic hash function [Barker and Kelsey 2015]. The program *init_hash_rng* in Figure 12a implements a simplified version of such a scheme.

When run, *init_hash_rng* generates a lazy hash function f for the key space $K = \{0, \ldots, \mathsf{MAX}\}$ for some fixed constant MAX. It also allocates a counter c as a reference initialized to 0. It returns a sampling function, let us call it h, that uses f and c to generate random Booleans. Each time h is called, it loads the current value n from c and hashes n with f to get a Boolean b. It then increments c and returns the Boolean b. Repeated calls to h return independent, uniformly sampled Booleans, so long as we make no more than MAX calls.

We prove that *init_hash_rng* is contextually equivalent to a "bounded" random number generator *init_bounded_rng* in Figure 12b that directly calls flip. The proof works by showing that, so long as $n \leq \mathsf{MAX}$, then each time a sample is generated, the value of n will not have been hashed before. Thus, we may couple the random hash value with the flip call in *init_bounded_rng*. This argument relies on the fact that the counter c is private, encapsulated state, which is easy to reason about using the relational judgment since Clutch is a separation logic.

6.4 Lazily Sampled Big Integers

Certain randomized data structures, such as treaps [Seidel and Aragon 1996], need to generate random *priorities* as operations are performed. One can view these priorities as an abstract data type equipped with a total order supporting two operations: (1) a *sample* function that randomly generates a new priority according to some distribution, and (2) a *comparison* operation that takes a pair of priorities (p_1, p_2) and returns -1 (if $p_1 < p_2$), 0 (if $p_1 = p_2$), or 1 (if $p_2 < p_1$). The full details of how priorities are used in such data structures are not relevant here. Instead, what is important to know is that it is ideal to avoid *collisions*, that is, sampling the same priority multiple times.

A simple way to implement priorities is to represent them as integers sampled from some fixed set $\{0, \ldots, n\}$. However, to minimize collisions, we may need to make n very large. But making n large has a cost, because then priorities requires more random bits to generate and more space to store. An alternative is to *lazily* sample the integer that represents the priority. Because we only need to compare priorities, we can delay sampling bits of the integer until they are needed to resolve ties during comparisons. A lazily-sampled integer can be encoded as a pair of a tape label ι and a linked list of length at most N, where each node in the list represents a *digit* of the integer in base B, with the head of the list being the most significant digit.

In the appendix [Gregersen et al. 2023b], we describe such an implementation of lazily-sampled integers, with $N = 8$ and $B = 2^{32}$. Our Coq development contains a proof that this implementation is contextually equivalent to code that eagerly samples a 256-bit integer by bit-shifting and adding 8 32-bit integers. Crucially, this contextual equivalence is at an *abstract* existential type τ. Specifically, we define the type of abstract priorities $\tau \triangleq \exists \alpha.\, (\mathsf{unit} \to \alpha) \times ((\alpha \times \alpha) \to \mathsf{int})$. Then we have the equivalence $\vdash (\mathit{sample\_lazy\_int}, \mathit{cmp\_lazy}) \simeq_{\mathrm{ctx}} (\mathit{sample256}, \mathit{cmp}) : \tau$ where *cmp* is just primitive integer comparison. The proof uses tapes to presample the bits of the lazy integer and couples these with the eager version. The *cmp_lazy* function traverses and mutates the linked lists representing the integers being compared, which separation logic is well-suited for reasoning about.

7 COUNTEREXAMPLES

This section justifies some design choices in Clutch by presenting counterexamples showing the unsoundness of two variants of the logic. In the first counterexample, we show that annotating sampling statements with tape labels is needed in our current formulation of the logic, since their omission leads to unsoundness. In the second, we show that combining prophecy variables [Jung et al. 2020] with the usual coupling rules of pRHL (without presampling) is unsound, implying that presampling cannot somehow be implemented in terms of prophecy variables.

7.1 Syntactic Restriction on Presampling

One may wonder whether it is necessary for tapes and labels to appear in the program and program state, but they do in fact play a subtle yet crucial role. Consider the following program *flip_or* that applies a logical disjunction to two fresh samples:

$$\begin{aligned} \mathit{flip\_or} \triangleq\ & \mathsf{let}\ x = \mathsf{flip}()\ \mathsf{in} \\ & \mathsf{let}\ y = \mathsf{flip}()\ \mathsf{in} \\ & x \mid\mid y \end{aligned}$$

and compare it to the program $\mathit{flip} \triangleq \mathsf{flip}()$ that just samples a bit. These two programs are obviously *not* contextually equivalent: with probability 3/4 the program *flip_or* will return true whereas the program *flip* only does so with probability 1/2. Yet, if we introduce a rule for flip that could draw from *any* presampling tape (*i.e.*, without requiring sampling statements to be annotated with the tape they will draw from), the logic would allow one to "prove" that they are equivalent.

Assume the following (unsound!) rule

$$\textsc{rel-tape-unsound}\quad \frac{\iota \hookrightarrow (1, b \cdot \vec{b}) \qquad \iota \hookrightarrow (1, \vec{b}) -\!\!* \Delta \vDash_{\mathcal{E}} K[b] \precsim e_2 : \tau}{\Delta \vDash_{\mathcal{E}} K[\,\mathsf{flip}()] \precsim e_2 : \tau}$$

that says that when sampling on the left-hand side, we may instead draw a bit b from *some* prover-chosen presampling tape ι. To see why this rule cannot be sound, we will show $\vDash \mathit{flip} \precsim \mathit{flip\_or} : \mathsf{bool}$.

First, we introduce two tapes with resources $\iota_1 \hookrightarrow (1, \epsilon)$ and $\iota_2 \hookrightarrow (1, \epsilon)$ on the left-hand side, either explicitly allocated in code as in Clutch or as pure ghost resources, if that is possible in our hypothetical logic. Second, we couple the tape ι_1 with the x-sampling and ι_2 with the y-sampling using REL-COUPLE-TAPE-L such that we end up with $\iota_1 \hookrightarrow (1, b_1)$ and $\iota_2 \hookrightarrow (1, b_2)$ and the goal $\vDash \mathsf{flip}() \precsim b_1 \mid\mid b_2 : \mathsf{bool}$. Finally, we do a case distinction on both b_1 and b_2: if both are true, or both are false, it does not matter which tape we use when applying REL-TAPE-UNSOUND. If, on the other hand, only b_i is true, we choose ι_i and apply REL-TAPE-UNSOUND which finishes the proof.

The crucial observation is that by labeling tapes in the program syntax, however, we prevent *the prover* from doing case analysis on presampled values to decide which tape to read—the syntax will dictate which tape to use and hence which value to read. Concretely, in $\mathbf{F}^{\text{rand}}_{\mu,\text{ref}}$, unlabeled flips always reduce uniformly at random and only labeled sampling statements will read from presampling tapes which prevents us from proving the unsound REL-TAPE-UNSOUND.

Besides motivating why soundly allowing presampling is subtle, this counterexample also emphasizes why the fact that labels appear in the program and in the program syntax is important. We do not claim that these annotations are absolutely necessary for some kind of presampling to be sound, as some very different formulation of the logic might be able to avoid them, but like for *prophecy variables* [Jung et al. 2020] where similar "ghost information" is needed in the actual program code, it is not obvious how to do without it. We remind the reader that presampling tapes nevertheless remain a proof-device as tapes can be erased through refinement as discussed in §2.

7.2 Incompatibility with Prophecy Variables

Presampling tapes bear *some* resemblance to prophecy variables in that they give us the means to talk about the future. However, prophecy variables, as previously developed in the context of Iris [Jung et al. 2020], are unsound for the (synchronous) coupling logic as illustrated below.

Assume the existence of two operators NewProph and Resolve p to b in our programming language and their (unsound for Clutch!) Hoare-triple specifications found below.

WP-NEWPROPH-UNSOUND

$$\{\mathsf{True}\}\ \mathsf{NewProph}\ \{p.\exists b \in \mathbb{B}.\, \mathsf{Proph}(p, b)\}$$

WP-RESOLVE-UNSOUND

$$\{\mathsf{Proph}(p, b) * b' \in \mathbb{B}\}\ \mathsf{Resolve}\, p\, \mathsf{to}\, b'\ \{b = b'\}$$

The specifications give us access to *Boolean one-shot prophecies* [Jung et al. 2020]. NewProph allocates a fresh prophecy variable p and a resource $\mathsf{Proph}(p, b)$ that tracks its future resolution b. Given ownership of $\mathsf{Proph}(p, b)$ then Resolve p to b' resolves the prophecy variable p to a value b' and knowledge that $b = b'$ was the case all along. To see why these operations and rules cannot be sound in the coupling logic, we will show $\vDash$ *flip_proph* $\precsim$ *flip* : bool where

$$
\begin{aligned}
\mathit{flip\_proph} \triangleq\ & \mathsf{let}\ p = \mathsf{NewProph}\ \mathsf{in} \\
& \mathsf{let}\ x = \mathsf{flip}()\ \mathsf{in} \\
& \mathsf{let}\ y = \mathsf{flip}()\ \mathsf{in} \\
& \mathsf{Resolve}\ p\ \mathsf{to}\ y; \\
& x\ \&\&\ y
\end{aligned}
$$

which cannot be the case as *flip_proph* returns true only with probability 1/4.

We unfold the relational judgment and apply WP-NEWPROPH-UNSOUND which gives us a prophecy about y and its future resolution b. If b is true, the evaluation on the left is predetermined to be $x \,\&\&\, \mathsf{true} = x$. By coupling the sampling of x with the flip() on the right using REL-COUPLE-RANDS, we finish using REL-RAND-L and WP-RESOLVE-UNSOUND. On the other hand, if b is false, the evaluation on the left is predetermined to be $x \,\&\&\, \mathsf{false} = \mathsf{false}$. We apply REL-RAND-L first and couple the sampling of y with the flip() on the right using REL-COUPLE-RANDS and finish using WP-RESOLVE-UNSOUND.

The counterexample shows that prophecy variables are unsound for the coupling logic, for the same reason that presampling is unsound without syntactic tape labels: If the prover can predict the outcomes of random samples ahead of time, it gives them too much power to choose which sampling they couple with.

8 COQ FORMALIZATION

All the results presented in the paper, including the background on probability theory, the formalization of the logic, and the case studies have been formalized in the Coq proof assistant [The Coq Development Team 2022]. The results about probability theory are built on top of the Coquelicot library [Boldo et al. 2015], extending their results to real series indexed by countable types.

Although we build our logic on top of Iris [Jung et al. 2018], significant work is involved in formalizing the operational semantics of probabilistic languages, our new notion of weakest precondition that internalizes the coupling-based reasoning, and the erasure theorem that allows us to conclude the existence of a coupling. Our development integrates smoothly with the Iris Proof Mode [Krebbers et al. 2017b] and we have adapted much of the tactical support from ReLoC [Frumin et al. 2021b] to reason about the relational judgment.

9 RELATED WORK

Separation logic. Relational separation logics have been developed on top of Iris for a range of properties, such as contextual refinement [Frumin et al. 2021b; Krebbers et al. 2017b; Timany and Birkedal 2019; Timany et al. 2018], simulation [Chajed et al. 2019; Gäher et al. 2022; Timany et al. 2021], and security [Frumin et al. 2021a; Georges et al. 2022; Gregersen et al. 2021]. The representation of the right-hand side program as a resource is a recurring idea, but our technical construction with run ahead is novel. With the exception of Tassarotti and Harper [2019], probabilistic languages have not been considered in Iris. Tassarotti and Harper develop a logic to show refinement between a probabilistic program and a semantic model, not a program. The logic relies on couplings, but it requires synchronization of sampling.

In Batz et al. [2019], a framework in which logical assertions are functions ranging over the non-negative reals is presented. The connectives of separation logic are given an interpretation as maps from pairs of non-negative reals to the positive reals. This work focuses on proving quantitative properties of a single program, *e.g.*, bounding the probability that certain events happen. A variety of works have developed separation logics in which the separating conjunction models various forms of probabilistic independence [Bao et al. 2021, 2022; Barthe et al. 2020]. For example, the statement $P * Q$ is taken to mean "the distribution of P is independent from the distribution of Q".

Prophecy variables [Abadi and Lamport 1988, 1991] have been integrated into separation logic in both unary [Jung et al. 2020] and relational settings [Frumin et al. 2021b]. The technical solution uses program annotations and physical state reminiscent of our construction with presampling tapes, but prophecy resolution is a physical program step, whereas presampling in our work is a logical operation. Prophecies can also be erased through refinement [Frumin et al. 2021b].

Probabilistic couplings. Probabilistic couplings are a technique from probability theory that can be used to prove equivalences between distributions or mixing times of Markov chains [Aldous 1983]. In computer science, they have been used to reason about relational properties of programs such as equivalences [Barthe et al. 2015] and differential privacy [Barthe et al. 2016a]. However, these logics requires the sampling points on both programs to be synchronized in order to construct couplings. In a higher-order setting, the logic by Aguirre et al. [2018] establish so-called "shift couplings" between probabilistic streams that evolve at different rates, but these rules are ad-hoc and limited to the stream type. Also in the higher-order setting, Aguirre et al. [2021] use couplings to reason about adversarially-defined properties, however they only support synchronous couplings, first-order global state, and use a graded state monad to enforce separation of adversary memories.

Logical relations. Step-indexed logical relations have been applied to reason about contextual equivalence of probabilistic programs in a variety of settings. Bizjak and Birkedal [2015] develop logical relations for a language similar to ours, although only with first-order state. This work has since been extended to a language with continuous probabilistic choice (but without state and impredicative polymorphism) [Wand et al. 2018], for which equivalence is shown by establishing a measure preserving transformation between the sources of randomness for both programs. Recently, this was further extended to support nested inference queries [Zhang and Amin 2022].

Another line of work [Dal Lago and Gavazzo 2021, 2022] uses so called differential logical relations to reason about contextual distance rather than equivalence. Programs are related using metrics rather than equivalence relations, which allows to quantify how similar programs are.

Cryptographic frameworks. CertiCrypt [Barthe et al. 2009, 2010] is a framework for cryptographic game-playing proofs written in a simple probabilistic first-order while-language ("pWhile"). CertiCrypt formalizes a denotational semantics for pWhile in Coq and supports reasoning about the induced notion of program equivalence via a pRHL, and provides dedicated tactics for lazy/eager sampling transformations. These kind of transformations are non-trivial for expressive languages like ours. CertiCrypt also provides a quantitative unary logic.

EasyCrypt [Barthe et al. 2013] is a standalone prover for higher-order logic building on CertiCrypt's ideas. It leverages the first-order nature of pWhile for proof automation via SMT solvers. EasyCrypt extends pWhile with a module system [Barbosa et al. 2021] to support reasoning about abstract code as module parameters. It integrates a quantitative unary logic with pRHL, and supports reasoning about complexity in terms of oracle calls [Barbosa et al. 2021]. Both automation and these kind of properties are out of scope for our work but would be interesting future directions.

In FCF [Petcher and Morrisett 2015], programs are written as Coq expressions in the free subdistributions monad. Proofs are conducted in a pRHL-like logic, where successive sampling statements can be swapped thanks to the commutativity of the monad.

SSProve [Abate et al. 2021; Haselwarter et al. 2021] supports modular crypto proofs by composing "packages" of programs written in the free monad for state and probabilities. The swap rule in SSProve allows exchanging commands which maintain a state invariant. Reasoning about dynamically allocated local state is not supported.

IPDL [Gancher et al. 2023] is a process calculus for stating and proving cryptographic observational equivalences. IPDL is mechanized in Coq and targeted at equational reasoning about interactive message-passing in high-level cryptographic protocol models, and hence considers a different set of language features.

10 CONCLUSION

We have presented Clutch, a novel higher-order probabilistic relational separation logic with support for asynchronous probabilistic coupling-based proofs of contextual refinement and equivalence of probabilistic higher-order programs with local state and impredicative polymorphism. We have proved the soundness of Clutch formally in Coq using a range of new technical concepts and ideas such as *left-partial couplings*, *presampling tapes*, and a *coupling modality*. We have demonstrated the usefulness of our approach through several example program equivalences that, to the best of our knowledge, were not possible to establish with previous methods.

DATA AVAILABILITY STATEMENT

The Coq formalization accompanying this work is available on Zenodo [Gregersen et al. 2023a] and on GitHub at https://github.com/logsem/clutch.

ACKNOWLEDGMENTS

This work was supported in part by a Villum Investigator grant (no. 25804), Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation. This work was co-funded by the European Union (ERC, CHORDS, 101096090). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them. This material is based upon work supported in part by the National Science Foundation under Grant No. 2318724.

REFERENCES

Martín Abadi and Leslie Lamport. 1988. The Existence of Refinement Mappings. In *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS '88), Edinburgh, Scotland, UK, July 5-8, 1988.* 165–175. https://doi.org/10.1109/LICS.1988.5115

Martín Abadi and Leslie Lamport. 1991. The Existence of Refinement Mappings. *Theor. Comput. Sci.* 82, 2 (1991), 253–284. https://doi.org/10.1016/0304-3975(91)90224-P

Carmine Abate, Philipp G. Haselwarter, Exequiel Rivas, Antoine Van Muylder, Théo Winterhalter, Catalin Hritcu, Kenji Maillard, and Bas Spitters. 2021. SSProve: A Foundational Framework for Modular Cryptographic Proofs in Coq. In *34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21-25, 2021.* 1–15. https://doi.org/10.1109/CSF51468.2021.00048

Alejandro Aguirre, Gilles Barthe, Lars Birkedal, Ales Bizjak, Marco Gaboardi, and Deepak Garg. 2018. Relational Reasoning for Markov Chains in a Probabilistic Guarded Lambda Calculus. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings.* 214–241. https://doi.org/10.1007/978-3-319-89884-1_8

Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg, Shin-ya Katsumata, and Tetsuya Sato. 2021. Higher-order probabilistic adversarial computations: categorical semantics and program logics. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–30. https://doi.org/10.1145/3473598

David J. Aldous. 1983. Random walks on finite groups and rapidly mixing Markov chains. *Séminaire de probabilités de Strasbourg* 17 (1983), 243–297. http://www.numdam.org/item/SPS_1983__17__243_0/

José Bacelar Almeida, Cécile Baritel-Ruet, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Alley Stoughton, and Pierre-Yves Strub. 2019. Machine-Checked Proofs for Cryptographic Standards: Indifferentiability of Sponge and Secure High-Assurance Implementations of SHA-3. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM, 1607–1622. https://doi.org/10.1145/3319535.3363211

Andrew W. Appel. 2001. Foundational Proof-Carrying Code. In *16th Annual IEEE Symposium on Logic in Computer Science, Boston, Massachusetts, USA, June 16-19, 2001, Proceedings.* 247–256. https://doi.org/10.1109/LICS.2001.932501

Jialu Bao, Simon Docherty, Justin Hsu, and Alexandra Silva. 2021. A Bunched Logic for Conditional Independence. In *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021.* IEEE, 1–14. https://doi.org/10.1109/LICS52264.2021.9470712

Jialu Bao, Marco Gaboardi, Justin Hsu, and Joseph Tassarotti. 2022. A separation logic for negative dependence. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–29. https://doi.org/10.1145/3498719

Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, and Pierre-Yves Strub. 2021. Mechanized Proofs of Adversarial Complexity and Application to Universal Composability. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi (Eds.). ACM, 2541–2563. https://doi.org/10.1145/3460120.3484548

Elaine B. Barker and John M. Kelsey. 2015. *Recommendation for Random Number Generation Using Deterministic Random Bit Generators.* National Institute of Standards and Technology. https://doi.org/10.6028/nist.sp.800-90ar1

Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. 2013. EasyCrypt: A Tutorial. In *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures.* 146–166. https://doi.org/10.1007/978-3-319-10082-1_6

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

Gilles Barthe, Thomas Espitau, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2018. Proving expected sensitivity of probabilistic programs. *Proc. ACM Program. Lang.* 2, POPL (2018), 57:1–57:29. https://doi.org/10.1145/3158145

Gilles Barthe, Noémie Fong, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2016a. Advanced Probabilistic Couplings for Differential Privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM, 55–67. https://doi.org/10.1145/2976749.2978391

Gilles Barthe, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2016b. Proving Differential Privacy via Probabilistic Couplings. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016.* 749–758. https://doi.org/10.1145/2933575.2934554

Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. 2009. Formal certification of code-based cryptographic proofs. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009.* 90–101. https://doi.org/10.1145/1480881.1480894

Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. 2010. Programming Language Techniques for Cryptographic Proofs. In *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings.* 115–130. https://doi.org/10.1007/978-3-642-14052-5_10

Gilles Barthe, Justin Hsu, and Kevin Liao. 2020. A probabilistic separation logic. *Proc. ACM Program. Lang.* 4, POPL (2020), 55:1–55:30. https://doi.org/10.1145/3371123

Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. 2012. Probabilistic relational reasoning for differential privacy. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012.* 97–110. https://doi.org/10.1145/2103656.2103670

Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Thomas Noll. 2019. Quantitative separation logic: a logic for reasoning about probabilistic pointer programs. *Proc. ACM Program. Lang.* 3, POPL (2019), 34:1–34:29. https://doi.org/10.1145/3290347

Mihir Bellare and Phillip Rogaway. 1993. Random Oracles are Practical: A Paradigm for Designing Efficient Protocols. In *CCS '93, Proceedings of the 1st ACM Conference on Computer and Communications Security, Fairfax, Virginia, USA, November 3-5, 1993.* 62–73. https://doi.org/10.1145/168588.168596

Mihir Bellare and Phillip Rogaway. 2004. Code-Based Game-Playing Proofs and the Security of Triple Encryption. Cryptology ePrint Archive, Paper 2004/331. https://eprint.iacr.org/2004/331 https://eprint.iacr.org/2004/331.

Mihir Bellare and Phillip Rogaway. 2006. The Security of Triple Encryption and a Framework for Code-Based Game-Playing Proofs. In *Advances in Cryptology - EUROCRYPT 2006*, Serge Vaudenay (Ed.). 409–426.

Aleš Bizjak. 2016. *On Semantics and Applications of Guarded Recursion.* Ph. D. Dissertation. Aarhus University.

Ales Bizjak and Lars Birkedal. 2015. Step-Indexed Logical Relations for Probability. In *Foundations of Software Science and Computation Structures - 18th International Conference, FoSSaCS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings.* 279–294. https://doi.org/10.1007/978-3-662-46678-0_18

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

Olivier Bousquet and André Elisseeff. 2002. Stability and Generalization. *J. Mach. Learn. Res.* 2 (mar 2002), 499–526. https://doi.org/10.1162/153244302760200704

Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. 2019. Verifying concurrent, crash-safe systems with Perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019.* 243–258. https://doi.org/10.1145/3341301.3359632

Ugo Dal Lago and Francesco Gavazzo. 2021. Differential logical relations, part II increments and derivatives. *Theor. Comput. Sci.* 895 (2021), 34–47. https://doi.org/10.1016/j.tcs.2021.09.027

Ugo Dal Lago and Francesco Gavazzo. 2022. Effectful program distancing. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–30. https://doi.org/10.1145/3498680

Derek Dreyer, Amal Ahmed, and Lars Birkedal. 2011. Logical Step-Indexed Logical Relations. *Log. Methods Comput. Sci.* 7, 2 (2011). https://doi.org/10.2168/LMCS-7(2:16)2011

Derek Dreyer, Georg Neis, and Lars Birkedal. 2012. The impact of higher-order state and control effects on local relational reasoning. *J. Funct. Program.* 22, 4-5 (2012), 477–528. https://doi.org/10.1017/S095679681200024X

Cynthia Dwork and Aaron Roth. 2013. The Algorithmic Foundations of Differential Privacy. *Foundations and Trends® in Theoretical Computer Science* 9, 3-4 (2013), 211–407. https://doi.org/10.1561/0400000042

Taher Elgamal. 1985. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Trans. Inf. Theory* 31, 4 (1985), 469–472. https://doi.org/10.1109/TIT.1985.1057074

Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2021a. Compositional Non-Interference for Fine-Grained Concurrent Programs. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021.* 1416–1433. https://doi.org/10.1109/SP40001.2021.00003

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

Lennard Gäher, Michael Sammler, Simon Spies, Ralf Jung, Hoang-Hai Dang, Robbert Krebbers, Jeehoon Kang, and Derek Dreyer. 2022. Simuliris: a separation logic framework for verifying concurrent program optimizations. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–31. https://doi.org/10.1145/3498689

Joshua Gancher, Kristina Sojakova, Xiong Fan, Elaine Shi, and Greg Morrisett. 2023. A Core Calculus for Equational Proofs of Cryptographic Protocols. *Proc. ACM Program. Lang.* 7, POPL, Article 30 (jan 2023), 27 pages. https://doi.org/10.1145/3571223

Aïna Linn Georges, Alix Trieu, and Lars Birkedal. 2022. Le temps des cerises: efficient temporal stack safety on capability machines using directed capabilities. *Proc. ACM Program. Lang.* 6, OOPSLA1 (2022), 1–30. https://doi.org/10.1145/3527318

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

Simon Oddershede Gregersen, Alejandro Aguirre, Philipp G. Haselwarter, Joseph Tassarotti, and Lars Birkedal. 2023a. *Asynchronous Probabilistic Couplings in Higher- Order Separation Logic - Coq Artifact.* https://doi.org/10.5281/zenodo.8424490

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

Simon Oddershede Gregersen, Johan Bay, Amin Timany, and Lars Birkedal. 2021. Mechanized logical relations for termination-insensitive noninterference. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–29. https://doi.org/10.1145/3434291

Philipp G. Haselwarter, Exequiel Rivas, Antoine Van Muylder, Théo Winterhalter, Carmine Abate, Nikolaj Sidorenco, Catalin Hritcu, Kenji Maillard, and Bas Spitters. 2021. SSProve: A Foundational Framework for Modular Cryptographic Proofs in Coq. Cryptology ePrint Archive, Paper 2021/397. https://eprint.iacr.org/2021/397 https://eprint.iacr.org/2021/397.

Patricia Johann, Alex Simpson, and Janis Voigtländer. 2010. A Generic Operational Metatheory for Algebraic Effects. In *Proceedings of the 25th Annual IEEE Symposium on Logic in Computer Science, LICS 2010, 11-14 July 2010, Edinburgh, United Kingdom.* 209–218. https://doi.org/10.1109/LICS.2010.29

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

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

Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. 2020. The future is ours: prophecy variables in separation logic. *Proc. ACM Program. Lang.* 4, POPL (2020), 45:1–45:32. https://doi.org/10.1145/3371113

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015.* 637–650. https://doi.org/10.1145/2676726.2676980

Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017a. The Essence of Higher-Order Concurrent Separation Logic. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings.* 696–723. https://doi.org/10.1007/978-3-662-54434-1_26

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017b. Interactive proofs in higher-order concurrent separation logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017.* 205–217. https://doi.org/10.1145/3009837.3009855

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

Arno Mittelbach and Marc Fischlin. 2021. *The Theory of Hash Functions and Random Oracles - An Approach to Modern Cryptography.* Springer. https://doi.org/10.1007/978-3-030-63287-8

Adam Petcher and Greg Morrisett. 2015. The Foundational Cryptography Framework. In *Principles of Security and Trust - 4th International Conference, POST 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings.* 53–72. https://doi.org/10.1007/978-3-662-46666-7_4

Andrew M. Pitts and Ian D. B. Stark. 1998. Operational Reasoning for Functions with Local State. In *Higher Order Operational Techniques in Semantics*, A. D. Gordon and A. M. Pitts (Eds.). Cambridge University Press, 227–273.

Mike Rosulek. 2020. *The Joy of Cryptography.* http://web.engr.oregonstate.edu/~rosulekm/crypto/

Davide Sangiorgi and Valeria Vignudelli. 2016. Environmental bisimulations for probabilistic higher-order languages. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016,* *St. Petersburg, FL, USA, January 20 - 22, 2016*. 595–607. https://doi.org/10.1145/2837614.2837651

Raimund Seidel and Cecilia R. Aragon. 1996. Randomized Search Trees. *Algorithmica* 16, 4/5 (1996), 464–497. https://doi.org/10.1007/BF01940876

Kasper Svendsen and Lars Birkedal. 2014. Impredicative Concurrent Abstract Predicates. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*. 149–168. https://doi.org/10.1007/978-3-642-54833-8_9

Joseph Tassarotti and Robert Harper. 2019. A separation logic for concurrent randomized programs. *Proc. ACM Program. Lang.* 3, POPL (2019), 64:1–64:30. https://doi.org/10.1145/3290377

The Coq Development Team. 2022. The Coq Proof Assistant. https://doi.org/10.5281/zenodo.7313584

The Iris Development Team. 2022. The Iris 4.0 Reference. https://plv.mpi-sws.org/iris/appendix-4.0.pdf

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

Amin Timany and Lars Birkedal. 2019. Mechanized relational verification of concurrent programs with continuations. *Proc. ACM Program. Lang.* 3, ICFP (2019), 105:1–105:28. https://doi.org/10.1145/3341709

Amin Timany, Simon Oddershede Gregersen, Léo Stefanesco, Léon Gondelman, Abel Nieto, and Lars Birkedal. 2021. Trillium: Unifying Refinement and Higher-Order Distributed Separation Logic. *CoRR* abs/2109.07863 (2021). arXiv:2109.07863 https://arxiv.org/abs/2109.07863

Amin Timany, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2022. A Logical Approach to Type Soundness. (2022). https://iris-project.org/pdfs/2022-submitted-logical-type-soundness.pdf Unpublished manuscript.

Amin Timany, Léo Stefanesco, Morten Krogh-Jespersen, and Lars Birkedal. 2018. A logical relation for monadic encapsulation of state: proving contextual equivalences in the presence of runST. *Proc. ACM Program. Lang.* 2, POPL (2018), 64:1–64:28. https://doi.org/10.1145/3158152

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

Aaron Joseph Turon, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer. 2013b. Logical relations for fine-grained concurrency. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*. 343–356. https://doi.org/10.1145/2429069.2429111

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

Mitchell Wand, Ryan Culpepper, Theophilos Giannakopoulos, and Andrew Cobb. 2018. Contextual equivalence for a probabilistic language with continuous random variables and recursion. *Proc. ACM Program. Lang.* 2, ICFP (2018), 87:1–87:30. https://doi.org/10.1145/3236782

Yizhou Zhang and Nada Amin. 2022. Reasoning about "reasoning about reasoning": semantics and contextual equivalence for probabilistic programs with nested queries and recursion. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–28. https://doi.org/10.1145/3498677

Received 2023-07-11; accepted 2023-11-07