Contextual Refinement of Higher-Order Concurrent Probabilistic Programs

KWING HEI LI, Aarhus University, Denmark
ALEJANDRO AGUIRRE, Aarhus University, Denmark
JOSEPH TASSAROTTI, New York University, USA
LARS BIRKEDAL, Aarhus University, Denmark

We present Foxtrot, the first higher-order separation logic for proving contextual refinement of higher-order concurrent probabilistic programs with higher-order local state. From a high level, Foxtrot inherits various concurrency reasoning principles from standard concurrent separation logic, e.g. invariants and ghost resources, and supports advanced probabilistic reasoning principles for reasoning about complex probability distributions induced by concurrent threads, e.g. tape presampling and induction by error amplification. The integration of these strong reasoning principles is highly *non-trivial* due to the combination of probability and concurrency in the language and the complexity of the Foxtrot model; the soundness of the logic relies on a version of the axiom of choice within the Iris logic, which is not used in earlier work on Iris-based logics. We demonstrate the expressiveness of Foxtrot on a wide range of examples, including the adversarial von Neumann coin and the randombytes_uniform function of the Sodium cryptography software library.

All results have been mechanized in the Rocq proof assistant and the Iris separation logic framework.

1 Introduction

Randomization and concurrency are, independently, two important and widely-used language features in the world of computing. The former is used to implement probabilistic data structures, randomized algorithms, and machine learning algorithms; the latter is utilized to increase program throughput and to model distributed systems and multi-core architecture. Together, the combination of these two features can be found in concurrent randomized algorithms [5, 30] and in cryptography protocols [15]. However, formal methods for reasoning about concurrent probabilistic programs are limited and not very expressive, in part because concurrency introduces nondeterminism, which in semantic models does not compose straightforwardly with probabilistic choice [39].

1.1 Motivating Example and Prior Work

Consider the following fImpl function, where rand N samples an integer uniformly from $\{0, \ldots, N\}$:

$$\mathsf{fImpl} \triangleq \lambda\, \_.\ \mathsf{let}\ (x, y) = (\mathsf{rand}\ 7 \mid\mid\mid \mathsf{rand}\ 31)\ \mathsf{in}\ \ x \cdot 32 + y$$

Though fImpl appears simple, it already exhibits the combination of both probability and concurrency. When we call fImpl, it spawns two concurrent threads, which generate a random 3-bit and 5-bit number, respectively. Afterwards, it returns the binary concatenation of the two numbers generated. Note that since the random samples of both threads are independent, this function behaves the same as one that directly samples a random 8-bit number. Hence to reason about its correctness, one might want to show that its behavior is the same as the following fSpec function.

$$\mathsf{fSpec} \triangleq \lambda\, \_.\ \mathsf{rand}\ 255$$

To formally argue that the two programs behave the same, we might want to prove that fImpl is *contextually equivalent* to fSpec. Intuitively, an expression e_1 is contextually equivalent to another expression e_2 at a type τ if no well-typed contexts C of ground type can distinguish them, denoted by $\emptyset \vdash e_1 \simeq_{\mathrm{ctx}} e_2 : \tau$. In other words, the behavior of a client program remains unchanged if we

Authors' Contact Information: Kwing Hei Li, Aarhus University, Aarhus, Denmark, hei.li@cs.au.dk; Alejandro Aguirre, Aarhus University, Aarhus, Denmark, alejandro@cs.au.dk; Joseph Tassarotti, New York University, New York, USA, jt4767@nyu.edu; Lars Birkedal, Aarhus University, Aarhus, Denmark, birkedal@cs.au.dk.

replace any occurrence of the sub-program e_1 with e_2. Contextual equivalence is defined as the symmetric interior of contextual refinement, denoted by $\emptyset \vdash e_1 \precsim_{\text{ctx}} e_2 : \tau$, which is roughly defined to mean that, for any context C the observable behaviour of $C[e_1]$ is *included* in the observable behaviour of $C[e_2]$. Proving contextual equivalence or refinement is generally a difficult problem as the statement quantifies over *all* possible contexts, including those that might use concurrency and probability as well. In fact, to the best of our knowledge, there is *no* prior work on verifying contextual equivalence of higher-order programs utilizing *both* probability and concurrency (see Figure 1 for an overview of a selection of prior work for proving contextual refinement of less expressive languages and §6 for a more detailed summary).

	Probability	Concurrency	Logic	Local State	Mechanized
Prob. λ-calculi [9, 11, 31]	●	○	○	○	○
Wand et al. [41]	●	○	○	○	○
Bizjak and Birkedal [7]	●	○	○	●	○
ReLoC [17]	○	●	●	●	●
Aguirre and Birkedal [3]	●	◐ (ND)	○	○	○
Clutch [18]	●	○	●	●	●
Approxis [19]	●	○	●	●	●
Foxtrot (this paper)	●	●	●	●	●

Fig. 1. Systems for proving contextual refinement (Logic = implemented as a logic, ND = only nondeterminism).

1.2 Contributions and Key Ideas

In this paper, we present *Foxtrot*, a higher-order probabilistic concurrent separation logic, which we use to build the *first* logical relations model for proving contextual refinement and equivalence of concurrent probabilistic programs. At a high level, the Foxtrot logic supports reasoning about contextual refinement of higher-order probabilistic concurrent imperative programs written in ConcRandML, an ML-style programming language with higher-order dynamically-allocated local state, discrete probabilistic sampling, and dynamically allocated concurrently running threads.

Since ConcRandML is probabilistic, the observable behaviour of a program refers to the *probability* of observing some behaviour, such as termination. Hence, by contextual refinement $\emptyset \vdash e_1 \precsim_{\text{ctx}} e_2 : \tau$, we mean that the (limit of the) probability of termination of $C[e_1]$ is below the (limit of the) probability of termination of $C[e_2]$. The precise definition can be found in §2.4; here it suffices to know that the operational semantics is defined using (probabilistic stateful) schedulers, which at every step of the execution decide which thread to execute next, and that the termination probability is obtained by taking the supremum over all possible schedulers (following the notion of *may-termination* in previous work [3, 6]).

The central new notion in Foxtrot to reason about contextual refinement is a *logical concurrent probabilistic refinement judgment*. The refinement judgment $\Delta \vDash e \precsim e' : \tau$ intuitively means that e refines e' at the type τ. The context Δ assigns a relational interpretation to all type variables in τ, but can be ignored for now. The logical refinement provides a sound method to prove contextual refinement, i.e. $\emptyset \vDash e_1 \precsim e_2 : \tau$ implies $\emptyset \vdash e_1 \precsim_{\text{ctx}} e_2 : \tau$. Following the logical approach to relational refinement [37], the refinement judgment is defined in the Foxtrot logic, which is in turn defined on top of the Iris base logic [21]. The Foxtrot program logic is phrased using Hoare triples $\{P\}\ e\ \{Q\}$, which are defined (using ghost state) with an eye for relational refinement, much as in earlier work [18, 19]. Rules in Foxtrot are often expressed in terms of Hoare triples, so to prove a refinement judgment, we often unfold its definition to the Hoare triple layer. We later see how the refinement judgment is defined with Hoare triples in §3.1.

Roughly speaking, Foxtrot inherits proof rules for concurrency (including *invariants* and *ghost resources*) from earlier relational logics for concurrency [17], and *coupling-based proof rules* for probabilistic reasoning from earlier relational logics for probability [18, 19]. In fact, these basic rules are sufficient to verify a wide range of concurrent probabilistic programs, especially those where the concurrency and probability features do not interact with each other in complex ways. However, some programs feature complex behaviors arising from the complicated interaction between probability and concurrency, e.g. the flmpl function from §1 where one has to reason about the joint probability distribution produced by sampling in two separate threads. To address these issues, Foxtrot includes more advanced probabilistic reasoning principles, including asynchronous couplings using so-called *presampling tapes* (first introduced in the Clutch logic [18]) and *fragmented couplings* for reasoning about rejection samplers [19]. Moreover, inspired by Eris [4] and Approxis [19], Foxtrot also supports induction by error amplification through the use of *error credit* resources. We explain these reasoning principles in §3, and we demonstrate their expressiveness in the concurrent setting by using them to reason about a range of challenging program refinements in §4, e.g. an adversarial variant of the von Neumann coin program and the implementation of the randombytes_uniform function in the open-source cryptography software library Sodium. These examples (even including the smaller examples in §3) are, to the best of our knowledge, beyond the scope of previous techniques, as they require complex reasoning principles, e.g. reasoning about higher-order functions, local state, and execution of unknown code.

In light of this rough overview, the reader may wonder if Foxtrot is a trivial combination of earlier logics and/or what, if anything, is challenging in its development? The key technical novelty and challenges lie in the *definition* of the program logic, specifically in the weakest precondition predicate used in the definition of Hoare triples, and the construction of the model of this logic, which ultimately allows us to prove the soundness theorem above. One of the key challenges is that intuitively, to show that e_1 refines e_2, for every possible scheduler for the left-hand side program, we have to exhibit a scheduler for the right-hand side program e_2, so that e_1 can be coupled with e_2, and this is done by "piecing together" schedulers for subexpressions (to validate compositional program logic rules). This is significantly different from earlier work on relational logics for concurrency without probabilities [17, 38], where one only has to construct a single *trace* for the right-hand side. Now, with probabilistic features, the challenge is that we have to compose *families* of schedulers (because of the probabilistic branching we get a family of schedulers). To do so, we end up having to use a version of the *axiom of choice* in the Iris logic which, to the best of our knowledge, has not been used in earlier work on Iris-based program logics. Indeed, Foxtrot's model is significantly more sophisticated than that of its predecessors, such as Clutch [18] and Approxis [19]. Luckily, users of the logic need not understand the model; it suffices to know that we have used the model to prove adequacy of the logic (Theorem 3.1), which intuitively states that proving a Hoare triple in the Foxtrot logic implies that the termination probability of a program is upper-bounded by that of another, and which is used to prove the soundness theorem above (Theorem 3.3).

We believe it is a *strong feature* of Foxtrot that, with the new definition of weakest precondition and model, we are in fact able to prove soundness of the many strong reasoning principles mentioned above. We note that this is not automatic (all the proof rules of Foxtrot have been proved sound from scratch) and not trivial; in fact, as we later emphasize in §3, one of the proof rules that one might expect to hold given earlier work [18, 19], namely "presampling on the right", is *not* sound in the concurrent setting of Foxtrot.

Contributions. In summary, we provide:

- The *first* higher-order separation logic, Foxtrot, for reasoning about contextual refinement of concurrent probabilistic programs.

- A rich set of coupling logic rules for advanced probabilistic reasoning, proven sound using the novel model of Foxtrot.
- A collection of case studies showcasing the expressiveness of Foxtrot, which are beyond the scope of all previous techniques.
- All results in the paper are mechanized in the Rocq proof assistant [34], using the Iris separation logic framework [20, 21, 23, 24] and Coquelicot real analysis library [8].

Outline. We begin by presenting the preliminaries for the rest of the paper (§2). We then detail the features of Foxtrot and demonstrate the use of these features through small examples (§3). Next, we show how Foxtrot can scale to verify larger and more involved case studies (§4). Afterwards, we explain how the semantic model of Foxtrot is constructed and proven sound (§5). We lastly conclude with a discussion of related and future work (§6, §7).

2 Preliminaries

In this section, we first recall useful facts from probability theory in §2.1. We then describe the syntax and semantics of our programming language ConcRandML in §2.2 and §2.3. Lastly in §2.4, we present the definition of contextual refinement we use in the concurrent probabilistic setting.

2.1 Probability Theory

A *subdistribution* (henceforth simply *distribution*) on a countable set A is a function $\mu : A \to [0,1]$ s.t. $\sum_{a \in A} \mu(a) \leq 1$. We write the collection of distributions on A as $\mathcal{D}(A)$. Distributions admit monadic operations, with $(\mathrm{ret} \colon A \to \mathcal{D}(A))$ defined as $\mathrm{ret}(a)(a') \triangleq \mathsf{if}\ a = a'\ \mathsf{then}\ 1\ \mathsf{else}\ 0$ and $(\mathrm{bind} \colon (A \to \mathcal{D}(B)) \times \mathcal{D}(A) \to \mathcal{D}(B))$ defined as $\mathrm{bind}(f,\mu)(b) \triangleq \sum_{a \in A} \mu(a) \cdot f(a)(b)$. We often use the notation $\mu \gg\!= f$ for $\mathrm{bind}(f,\mu)$.

Some examples of distributions include the *null distribution* $\mathbf{0} : \mathcal{D}(A)$, given by $\lambda x.0$, and the uniform distribution $\mathfrak{U}N \colon \mathcal{D}(\mathbb{N})$, which returns $1/(N+1)$ for every $n \in \{0, \ldots, N\}$ and 0 otherwise.

The *expected value* of $X \colon A \to [0,1]$ w.r.t. μ is defined as $\mathbb{E}_\mu[X] \triangleq \sum_{a \in A} \mu(a) \cdot X(a)$. The *mass* of μ is defined as $|\mu| \triangleq \mathbb{E}_\mu[\lambda x.1]$.

Foxtrot uses *approximate* probabilistic coupling [19, 32] to relate pairs of probabilistic programs.

Definition 2.1 (Approximate Coupling). *Let $\mu_1 \in \mathcal{D}(A)$ and $\mu_2 \in \mathcal{D}(B)$. Given $\varepsilon \in [0,1]$ and a relation $R \subseteq A \times B$, we say that there exists an (ε, R)-coupling of μ_1 and μ_2 if for all $[0,1]$-valued random variables $X : A \to [0,1]$ and $Y : B \to [0,1]$, s.t. $(a,b) \in R$ implies $X(a) \leq Y(b)$, we have the inequality $\mathbb{E}_{\mu_1}[X] \leq \mathbb{E}_{\mu_2}[Y] + \varepsilon$. We write $\mu_1 \lesssim_\varepsilon \mu_2 : R$ if such an (ε, R)-coupling exists.*

2.2 The ConcRandML Language

Our examples are written in the ConcRandML language, a higher-order ML-style programming language with higher-order local state, extended with discrete probabilistic sampling and fork-based unstructured concurrency. The syntax of the language is defined by the grammar below:

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

The syntax is mostly standard, e.g. $\mathsf{ref}\, e$, $!\, e$, and $e_1 \leftarrow e_2$ respectively allocate, read from, and write onto a reference. The expression $\mathsf{rand}\, N$ samples from $\mathfrak{U}N$, and $\mathsf{fork}\, e$ spawns a new thread with

e. ConcRandML provides various *atomic* operations for synchronizing threads, e.g. fetch-and-add faa e_1 e_2 adds the integer e_2 to the value v stored at location e_1 and returns v.

The state σ is given by a pair of maps. The first map models the heap as a finite map from locations to values. The second map is used for keeping track of *presampling tapes*, the discussion of which is postponed until §3.3. A program configuration $\rho \in \mathit{List}(\mathit{Expr}) \times \mathit{State}$ is given by a pair containing the list of currently executing threads and the state. We say a configuration is *final* if the first expression in the thread list is a value.

To define contextual refinement of ConcRandML programs (§2.4), we uses its typing system, which follows standard ML-style rules. Typing is expressed as $\Theta \mid \Gamma \vdash e : \tau$ where Γ is a context assigning types to program variables, and Θ contains type variables that may occur in Γ and τ.

2.3 Operational Semantics

The operational semantics of ConcRandML programs is defined in 3 steps: (1) a step function for stepping a single expression, (2) a stepping function for a configuration w.r.t. a scheduler, and (3) full program execution w.r.t. a scheduler[1].

Expressions. We consider a mostly standard call-by-value semantics. The step : $(\mathit{Expr}, \mathit{State}) \to \mathcal{D}(\mathit{Expr}, \mathit{State}, \mathit{List}(\mathit{Expr}))$ function takes an expression and a state, and returns a distribution over a new expression, state, and a (possibly empty) list of newly spawned threads. We display some reductions below; in particular rand N steps to an integer between 0 and N uniformly at random.

$$\begin{aligned}
\mathsf{step}(\text{if true then } e_1 \text{ else } e_2, \sigma) &= \mathsf{ret}(e_1, \sigma, [])\\
\mathsf{step}(\text{fork } e, \sigma) &= \mathsf{ret}((), \sigma, [e])\\
\mathsf{step}(\text{rand } N, \sigma) &= \lambda\,(n, \sigma, [])\,.\ \tfrac{1}{N+1} \text{ if } n \in \{0, \ldots, N\} \quad \text{and } 0 \text{ otherwise}
\end{aligned}$$

Thread Pools and Schedulers. The tpStep function takes a configuration $\rho = (\vec{e}, \sigma)$ and an index j and returns a distribution over new configurations after stepping the j-th thread:

$$\mathsf{tpStep}(\vec{e}, \sigma)(j) \triangleq \begin{cases} \mathbf{0} & \text{if } (\vec{e}, \sigma) \text{ is final,}\\ \mathsf{ret}(\vec{e}, \sigma) & \text{if } e_j \in \mathit{Val} \text{ or } j \geq |\vec{e}|,\\ \mathsf{step}(e_j, \sigma) \gg\!= \lambda\,(e_j', \sigma', \vec{e}').\ \mathsf{ret}(\vec{e}[j \mapsto e_j'] +\!\!+ \vec{e}', \sigma') & \text{otherwise.} \end{cases}$$

If ρ is final, we return immediately. If e_j is a value or does not exist, we take a stutter step. Otherwise, we step the j-th thread and add the spawned threads to the thread pool.

The choice of which thread to step is determined by a *scheduler*. A (probabilistic, stateful) scheduler is defined by a transition function $\zeta : (\mathit{SchedSt} \times \mathit{Cfg}) \to \mathcal{D}(\mathit{SchedSt} \times \mathbb{N})$, which takes in an internal state $\Xi \in \mathit{SchedSt}$ and a configuration ρ, and returns a distribution on its new internal state and the index of the thread to step next.

Given a configuration ρ, a scheduler ζ, and a scheduler state Ξ, we can now define the single scheduler-step reduction function $\mathsf{schStep}_\zeta(\Xi, \rho) \in \mathcal{D}(\mathit{SchedSt} \times \mathit{Cfg})$ as follows:

$$\mathsf{schStep}_\zeta(\Xi, \rho) \triangleq \zeta(\Xi, \rho) \gg\!= \lambda\,(\Xi', j).\ \mathsf{tpStep}(\rho, j) \gg\!= \lambda\,\rho'.\ \mathsf{ret}(\Xi', \rho')$$

Informally, $\mathsf{schStep}_\zeta(\Xi, \rho)$ first evaluates $\zeta(\Xi, \rho)$ to get a new state Ξ' and index j, steps the j-th thread to obtain the new configuration ρ', and returns the new scheduler state and configuration.

[1] ConcRandML is the same language studied in Coneris [26] and we refer readers to that paper for more details.

Full Program Execution. We now define n-step program execution w.r.t. a scheduler ζ with the recursive function $\mathsf{exec}_{\zeta,n} : (\mathit{SchedSt} \times \mathit{Cfg}) \to \mathcal{D}(\mathit{Val})$ defined below. Intuitively, $\mathsf{exec}_{\zeta,n}(\Xi, \rho)(v)$ represents the probability of returning a value v in the first thread after at most n steps of ρ w.r.t. the scheduler ζ with starting scheduler state Ξ. The full program execution is then defined as the limit of $\mathsf{exec}_{\zeta,n}$, which exists by monotonicity and continuity, i.e. $\mathsf{exec}_\zeta(\Xi, \rho) \triangleq \lim_{n\to\infty} \mathsf{exec}_{\zeta,n}(\Xi, \rho)$.

$$\mathsf{exec}_{\zeta,n}(\Xi, \rho) \triangleq \begin{cases} \mathsf{ret}\, v & \text{if } \rho = (v \cdot \vec{e}, \sigma) \text{ for some } v \in \mathit{Val}, \\ \mathbf{0} & \text{if } n = 0 \text{ and } \rho \text{ is not final}, \\ \mathsf{schStep}_\zeta(\Xi, \rho) \gg\!= \mathsf{exec}_{\zeta,n-1} & \text{otherwise.} \end{cases}$$

The probability that a starting configuration ρ terminates under scheduler ζ with starting scheduler state Ξ is hence defined as $\mathsf{exec}^{\Downarrow}_\zeta(\Xi, \rho) \triangleq |\mathsf{exec}_\zeta(\Xi, \rho)|$.

The upper termination probability of an arbitrary configuration ρ is taken as the supermum of the termination probability ranging over all possible schedulers and initial scheduler states, i.e. $\mathsf{exec}^{\sqcup\Downarrow}(\rho) \triangleq \sup_{\zeta,\Xi} \mathsf{exec}^{\Downarrow}_\zeta(\Xi, \rho)$. We use the upper termination probability $\mathsf{exec}^{\sqcup\Downarrow}$ as the observation predicate for contextual refinement. In other words, we consider *may-termination* [3, 6] as the main property of interest for the refinement[2].

We here also provide an auxiliary definition of partial program execution $\mathsf{pexec}_{\zeta,n} : (\mathit{SchedSt} \times \mathit{Cfg}) \to \mathcal{D}(\mathit{SchedSt} \times \mathit{Cfg})$, used in the model of Foxtrot.

$$\mathsf{pexec}_{\zeta,n}(\Xi, \rho) = \begin{cases} \mathsf{ret}(\Xi, \rho) & \text{if } \rho \text{ is final or } n = 0, \\ \mathsf{schStep}_\zeta(\Xi, \rho) \gg\!= \mathsf{pexec}_{\zeta,n-1} & \text{otherwise.} \end{cases}$$

We can view pexec as a relaxation of exec which keeps probability mass on configurations that are not final, whereas the latter only considers final configurations.

2.4 Contextual Refinement

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 where $\Theta \mid \Gamma \vdash e : \tau$ 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, the upper termination probability of $C[e_1]$ is bounded by the upper termination probability of $C[e_2]$:

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

Note that the statement itself is in the meta-logic (*e.g.*, Rocq) and makes no mention of Foxtrot or Iris. Contextual equivalence $\Theta \mid \Gamma \vdash e_1 \simeq_{\mathrm{ctx}} e_2 : \tau$ is defined as a refinement in both directions, *i.e.* $(\Theta \mid \Gamma \vdash e_1 \precsim_{\mathrm{ctx}} e_2 : \tau) \wedge (\Theta \mid \Gamma \vdash e_2 \precsim_{\mathrm{ctx}} e_1 : \tau)$. We simply write $e_1 \precsim_{\mathrm{ctx}} e_2$ or $e_1 \simeq_{\mathrm{ctx}} e_2$ if the type τ can be inferred and that Γ and Θ are empty.

3 Logic

Now we introduce the rules of the Foxtrot logic. We first present a small set of its propositions:

$$P, Q \in \mathit{iProp} ::= \mathsf{True} \mid \mathsf{False} \mid P \wedge Q \mid P \vee Q \mid P \Rightarrow Q \mid \forall x.\, P \mid \exists x.\, P \mid \\ P * Q \mid P \mathrel{-\!\!*} Q \mid \triangleright P \mid \boxed{P}^{\iota} \mid \boxed{\overline{g}}^{\gamma} \mid {}_{\mathcal{E}_1}\!\Rightarrow\!\!\!\Rightarrow_{\mathcal{E}_2} P \mid j \Rightarrow e \mid \ell \mapsto v \mid \ell \mapsto_{\mathsf{s}} v \mid \\ \{P\}\, e\, \{Q\}_{\mathcal{E}} \mid \rightsquigarrow P \mid \kappa \hookrightarrow (N, \vec{n}) \mid \kappa \hookrightarrow_{\mathsf{s}} (N, \vec{n}) \mid \lightning(\varepsilon) \mid \ldots$$

[2]We leave the problem of *must-termination* refinement of concurrent probabilistic programs for future work (see §7).

Foxtrot is built on top of the Iris base logic [23] and it inherits many of the basic propositions found in standard Iris separation logics, such as the *later* modality $\triangleright$, *invariants* $\boxed{I}^{\iota}$, *ghost resources* $\ulcorner g \urcorner^{\gamma}$, and *fancy updates* ${}_{\mathcal{E}_1}\!\Rightarrow_{\mathcal{E}_2} P$. The specification program on the right-hand side is represented by the specification thread connective $j \Rightarrow e$, which intuitively states that the j-th right-hand side thread is currently executing expression e. We use the points-to connectives $\ell \mapsto v$ and $\ell \mapsto_{\mathrm{s}} v$ to represent exclusive ownership of the location ℓ storing value v in the left-hand side program and right-hand side program, respectively.

Central to Foxtrot's logic is the Hoare triple proposition[3], and we give meaning to the Hoare triple via the adequacy theorem Theorem 3.1:

Theorem 3.1 (Adequacy). *If we can prove* $\{0 \Rightarrow e'\}\ e\ \{v.\exists v'.\ 0 \Rightarrow v'\}$ *in Foxtrot, then for all states* σ, σ'*, we have* $\mathsf{exec}^{\sqcup\Downarrow}([e], \sigma) \leq \mathsf{exec}^{\sqcup\Downarrow}([e'], \sigma')$.

Roughly speaking, the theorem states that if we are able to prove the Hoare triple for the expression e assuming that the 0-th thread on the right stores e' initially, and that the execution of that same thread eventually returns a value, then the upper termination probability of e is upper-bounded by that of e'.

Foxtrot also supports other propositions for advanced probabilistic reasoning such as the *probabilistic update modality* $\leadsto P$, *presampling-tape* connectives $\kappa \hookrightarrow (N, \vec{n})$ and $\kappa \hookrightarrow_{\mathrm{s}} (N, \vec{n})$, as well as *error credits* $\lightning(\varepsilon)$. We describe these in more details later in this section.

In §3.1, we first present the logical relation that gives rise to the refinement judgment introduced in §1.2, as well as the key results of the refinement judgment. We then present an overview of the rules of Foxtrot, from basic program-logic rules in §3.2 to more advanced rules for more complicated couplings in §3.3. We emphasize that though various rules and features are inspired by prior work, as mentioned in §1.2, proving all the rules are sound in our novel Foxtrot model is highly *non-trivial* and is *not automatic*, which we explain more in detail in §5.

3.1 Logical Relations

In §1.2, we introduced the logical concurrent probabilistic refinement judgment for proving contextual refinement of ConcRandML programs. In Foxtrot, this judgment $\Delta \vDash e \precsim e' : \tau$ is defined as $\forall K,\ j.\ \{j \Rightarrow K[e']\}\ e\ \{v.\exists v'.\ j \Rightarrow K[v'] * [\![\tau]\!]_\Delta(v, v')\}$, where K quantifies over all evaluation contexts. Here $[\![\tau]\!]_\Delta$ represents the semantic interpretation of type τ defined inductively on the type, which intuitively describes when two values of a certain type are considered related. The way we define the logical relations and encode it within the logic is standard and similar to that in previous work [37]. We extend the refinement judgment to open terms in the standard way, *i.e.* $\Delta \mid \Gamma \vDash e \precsim e' : \tau \triangleq \forall \gamma, \gamma'.[\![\Gamma]\!]_\Delta(\gamma, \gamma') \mathrel{-\!\!*} \Delta \vDash e\gamma \precsim e'\gamma' : \tau$, and prove the fundamental lemma of the binary logical relations:

Lemma 3.2 (Fundamental Lemma of Binary Logical Relations). *If* $\Theta \mid \Gamma \vdash e : \tau$ *then for all* Δ *assigning a relational interpretation to every type variable in* Θ*, we have* $\Delta \mid \Gamma \vDash e \precsim e : \tau$.

We then show that the refinement relation is sound with respect to contextual refinement:

Theorem 3.3 (Soundness). *Let* Θ *be a type variable context, and assume that, for all* Δ *assigning a relational interpretation to all type variables in* Θ*, we can derive* $\Delta \mid \Gamma \vDash e_1 \precsim e_2 : \tau$*. Then* $\Theta \mid \Gamma \vdash e_1 \precsim_{ctx} e_2 : \tau$.

In addition to the binary refinement relations, we also additionally construct a *unary* logical relations model for proving type safety of ConcRandML programs where $\Delta \vDash e : \tau \triangleq$

[3]Hoare triples are annotated with a mask $\mathcal{E}$ and we omit it if it is the top mask $\top$.

$\{\mathsf{True}\}\ e\ \{v.[\![\tau]\!]_\Delta(v)\}$. This is then similarly extended to open terms: $\Delta \mid \Gamma \vDash e : \tau \triangleq \forall \gamma.[\![\Gamma]\!]_\Delta(\gamma) \mathrel{-\!\!*} \Delta \vDash e\gamma : \tau$ and we prove its expected fundamental lemma, which is used later in the adversarial von Neumann coin example in §4.1 to reason about unknown adversaries.

Lemma 3.4 (Fundamental Lemma of Unary Logical Relations). *If* $\Theta \mid \Gamma \vdash e : \tau$ *then for all* Δ *assigning a unary interpretation to all type variables in* Θ*, we have* $\Delta \mid \Gamma \vDash e : \tau$.

3.2 Basic Rules

In this subsection, we cover structural rules, rules for symbolically executing the left or right-hand side programs, and the standard rule for coupling programs.

Standard Iris Rules. Foxtrot inherits many standard rules in similar Iris separation logics, such as HT-BIND for reasoning about sequences of instructions and the frame rule HT-FRAME for framing resources around a Hoare triple. We symbolically execute the left-hand side program with familiar-looking rules such as that of HT-LOAD and HT-FORK. As mentioned in §1.2, Foxtrot is a concurrent separation logic and it supports invariants (with rules such as HT-INV-ALLOC and HT-INV-OPEN) and ghost resources (standard rules omitted here for brevity).

$$\text{HT-BIND}\ \frac{\{P\}\ e\ \{v.Q\} \qquad \forall v.\ \{Q\}\ K[v]\ \{R\}}{\{P\}\ K[e]\ \{R\}} \qquad \text{HT-FRAME}\ \frac{\{P\}\ e\ \{Q\}}{\{P * R\}\ e\ \{Q * R\}} \qquad \text{HT-LOAD}\ \frac{}{\{l \mapsto v\}\ !l\ \{w.w = v * l \mapsto v\}}$$

$$\text{HT-FORK}\ \frac{\{\mathsf{True}\}\ e\ \{\mathsf{True}\}}{\{\mathsf{True}\}\ \mathsf{fork}\ e\ \{w.w = ()\}} \qquad \text{HT-INV-ALLOC}\ \frac{\{\boxed{P}^\iota * Q\}\ e\ \{R\}}{\{P * Q\}\ e\ \{R\}} \qquad \text{HT-INV-OPEN}\ \frac{e\ \text{atomic} \qquad \{\triangleright I * P\}\ e\ \{\triangleright I * Q\}_{\mathcal{E}}}{\{\boxed{I}^\iota * P\}\ e\ \{Q\}_{\mathcal{E} \uplus \{\iota\}}}$$

We implement the parallel composition operator with the fork primitive, and with ghost resources, we prove the familiar-looking rule HT-PAR-COMP for parallel composition:

$$\frac{\{P_1\}\ e_1\ \{v_1.\ Q_1\ v_1\} \qquad \{P_2\}\ e_2\ \{v_2.\ Q_2\ v_2\}}{\{P_1 * P_2\}\ e_1 \mid\mid\mid e_2\ \{(v_1, v_2).\ Q_1\ v_1 * Q_2\ v_2\}}\ \text{HT-PAR-COMP}$$

Probabilistic Update Modality. We previously saw that we can symbolically execute the left-hand side program with the Hoare triple directly. Naturally, one might ask, how can we do so for the right? The answer is that we do so with the *probabilistic update modality* $\leadsto$, a construct first introduced in Coneris [26] for reasoning about randomized logical atomicity. Here, we extend the modality for manipulating the right-hand side program; we also later see how the probabilistic update modality is used in more complicated rules, such as those for presampling onto tapes and for generating error credits.

The probabilistic update modality supports monadic-like rules, i.e. PUPD-RET and PUPD-BIND. Most importantly, the probabilistic update modality can be eliminated in the precondition of a Hoare triple (HT-PUPD-ELIM). As a consequence, rules for symbolically executing the right-hand side program can be succinctly expressed with the probabilistic update modality, which is different in presentation from earlier relational logics [17, 37]. For example, PUPD-LOAD and PUPD-FORK symbolically steps a load and fork expression on the right-hand side program, respectively.

$$\frac{P}{\leadsto P}\ \text{PUPD-RET} \qquad \frac{\leadsto P \qquad P \mathrel{-\!\!*} \leadsto Q}{\leadsto Q}\ \text{PUPD-BIND} \qquad \frac{\{P * Q\}\ e\ \{R\}}{\{(\leadsto P) * Q\}\ e\ \{R\}}\ \text{HT-PUPD-ELIM}$$

$$\frac{l \mapsto_{\mathrm{s}} v \qquad j \Rightarrow K[!\,l]}{\leadsto (l \mapsto_{\mathrm{s}} v * j \Rightarrow K[v])}\ \text{PUPD-LOAD} \qquad \frac{j \Rightarrow K[\,\mathsf{fork}\ e]}{\leadsto (j \Rightarrow K[()] * \exists j'.\ j' \Rightarrow e)}\ \text{PUPD-FORK}$$

In addition, Foxtrot provides PUPD-PAR-COMP for parallel composition on the specification side:

$$\frac{j \Rightarrow K[e_1 \,|||\, e_2]}{\Rrightarrow \exists j_1, j_2, K_1, K_2.\ \left(\begin{array}{l} j_1 \Rightarrow K_1[e_1] * j_2 \Rightarrow K_2[e_2] * \\ \forall v_1, v_2.\ j_1 \Rightarrow K_1[v_1] \mathrel{-\!\!*} j_2 \Rightarrow K_2[v_2] \mathrel{-\!\!*} \Rrightarrow j \Rightarrow K[(v_1, v_2)] \end{array}\right)} \text{PUPD-PAR-COMP}$$

Couplings. Foxtrot makes use of (approximate) *probabilistic couplings* [27, 35] to relate the randomness of two programs. We present the following simple yet useful coupling rule HT-COUPLE, which allows us to reason as if the two sampled values on both programs are related by a bijective function f under the same sampling space $\{0, \ldots, N\}$.

$$\frac{f \text{ bijection} \qquad \forall n \le N.\ \{j \Rightarrow K[f(n)]\}\ n\ \{\Phi\}}{\{j \Rightarrow K[\,\mathsf{rand}\,N]\}\ \mathsf{rand}\,N\ \{\Phi\}} \text{HT-COUPLE}$$

Together with the other logical facilities for reasoning about concurrency, e.g. invariants, the coupling rule HT-COUPLE is sufficient for reasoning about various concurrent probabilistic programs. As an example, consider the left-hand side entropy-mixer program in Figure 2, originally studied in Probabilistic Concurrent Outcome Logic [43]. Here, we are given two sources of entropy, x_1 and x_2, the former being unreliable as it can be adversarially controlled by the scheduler, while x_2 is of high quality. Nonetheless, we are still able to "combine" these two sources and produce a high-quality output that behaves like a rand 1. We can verify this claim by showing contextual equivalence between the entropy-mixer program and a rand 1 program with Foxtrot. Even though this program uses both concurrency (parallel composition) and random sampling (rand), the bulk of the proof simply boils down to applying the soundness theorem (Theorem 3.3), defining an invariant to share the reference y between the concurrent threads, and applying HT-COUPLE with the right bijective function f.

$$\begin{array}{l} \mathsf{let}\ (y, r) = (\mathsf{ref}\ 0, \mathsf{ref}\ 0)\ \mathsf{in} \\ \left(\begin{array}{l} \mathsf{let}\ x_1 = !\,y\ \mathsf{in} \\ \mathsf{let}\ x_2 = \mathsf{rand}\ 1\ \mathsf{in} \\ r \leftarrow ((x_1 + x_2) \bmod 2) \end{array} \quad ||| \quad y \leftarrow 1\right) \\ !\,r \end{array} \quad \simeq_{\mathrm{ctx}} \quad \mathsf{rand}\ 1$$

Fig. 2. Entropy mixer example.

We summarize this subsection with the following slogan:

Slogan 1: Invariants, ghost resources, and HT-COUPLE are (almost) all you need.

3.3 Advanced Rules

Previously, we saw the standard coupling rule HT-COUPLE for coupling a rand N on the left with one on the right under a bijective function. However, unsurprisingly, there are times where it is not the case that there is conveniently a single rand of the same range on both programs. In those cases, Foxtrot uses other advanced logical facilities and rules to establish the couplings.

Combining Samplings in Separate Threads. Foxtrot supports rules for reasoning about programs that perform random samplings across different threads. Consider the batch sampling program batchImpl in Figure 3, which is a generalization of flmpl from §1. It computes rand N and rand M concurrently and returns the sum of the result with the first value multiplied by $(M + 1)$. In other words, it aggregates two parallelized random samples to produce another one. We can

show that this function returns a number that is uniformly distributed by showing its contextual equivalence to the function batchSpec that directly samples from rand$((N+1)\cdot(M+1)-1)$.

$$\begin{array}{l}\mathsf{batchImpl} \triangleq \\ \lambda\_.\ \mathsf{let}\ (x,y) = (\mathsf{rand}\ N \mid\mid\mid \mathsf{rand}\ M)\ \mathsf{in}\ \ x\cdot(M+1)+y\end{array} \simeq_{\mathrm{ctx}} \begin{array}{l}\mathsf{batchSpec} \triangleq \\ \lambda\_.\ \mathsf{rand}((N+1)\cdot(M+1)-1)\end{array}$$

Fig. 3. Batch-sampling example.

Let us first consider the left-to-right refinement first. To reason about random samplings occurring in different threads in batchImpl, Foxtrot uses presampling tapes (first introduced in Clutch [18]) to "compress" multiple samplings and reason about them as if they are one.

$$v \in \mathit{Val} ::= \ldots \mid \kappa \in \mathit{Label} \qquad \sigma \in \mathit{State} \triangleq (\mathit{Loc} \xrightarrow{\mathsf{fin}} \mathit{Val}) \times (\mathit{Label} \xrightarrow{\mathsf{fin}} \mathit{Tape})$$

$$e \in \mathit{Expr} ::= \ldots \mid \mathsf{tape}\ e \mid \mathsf{rand}\ e_1\ e_2 \qquad t \in \mathit{Tape} \triangleq \{(N,\vec{n}) \mid N \in \mathbb{N} \wedge \vec{n} \in \mathbb{N}^*_{\leq N}\}$$

As mentioned previously in §2.2, the state of our language contains a finite map from tape labels to presampling tapes. We also introduce two new constructs in ConcRandML, tape e and rand $e_1\ e_2$. The expression tape N allocates an empty tape and returns a new tape label (1). In addition to the regular unlabelled random sampling we have shown before, we can perform a *labelled* random sampling by supplying a tape label to the rand operation. If the tape label points to an empty tape, then the labelled random sampling instruction acts like a normal unlabelled one (2), *i.e.* it samples a number uniformly from the range. However, if it is not empty, we pop and return the first number in the tape *deterministically* (3). Perhaps surprisingly, there are *no* constructs in ConcRandML for presampling elements onto tapes, and so, tapes remain empty throughout the execution of the program (and hence, (3) is never applied normally). Instead, the tape presampling action is done only within the logic during the proof.

$$\mathrm{step}(\mathsf{tape}\ N, \sigma) = \mathrm{ret}(\kappa, \sigma[\kappa := (N,\epsilon)], []) \quad (\text{where } \kappa \text{ is fresh w.r.t. } \sigma) \tag{1}$$

$$\mathrm{step}(\mathsf{rand}\ \kappa\ N, \sigma) = \lambda\,(n,\sigma,[])\,.\ \mathsf{if}\ 0 \leq n \leq N\ \mathsf{then}\ \tfrac{1}{N+1}\ \mathsf{else}\ 0 \quad (\text{where } \sigma[\kappa] = (N,\epsilon)) \tag{2}$$

$$\mathrm{step}(\mathsf{rand}\ \kappa\ N, \sigma) = \mathrm{ret}(n, \sigma[\kappa := \vec{n}], []) \quad (\text{where } \sigma[\kappa] = (N, n\cdot\vec{n})) \tag{3}$$

Just like we have the heap points-to connectives on both sides of the refinement, we have the tapes connectives for both sides as well: $\kappa \hookrightarrow (N,\vec{n})$ and $\kappa \hookrightarrow_{\mathrm{s}} (N,\vec{n})$. The rules HT-ALLOC-TAPE and PUPD-ALLOC-TAPE allocates a new empty tape, and HT-RAND-TAPE returns the first element of a tape deterministically for a labelled rand on the left-hand side program. There is however *no* counterpart rule of HT-RAND-TAPE for the right-hand side program, as presampling onto tapes on the specification side is unsound.

HT-ALLOC-TAPE

$$\frac{}{\{\mathsf{True}\}\ \mathsf{tape}\ N\ \{\kappa.\ \kappa \hookrightarrow (N,\epsilon)\}}$$

PUPD-ALLOC-TAPE

$$\frac{j \Rightarrow K[\ \mathsf{tape}\ N]}{\Rrightarrow \exists\kappa.\ j \Rightarrow K[\kappa] * \kappa \hookrightarrow_{\mathrm{s}} (N,\epsilon)}$$

HT-RAND-TAPE

$$\frac{}{\{\kappa \hookrightarrow (N, n\cdot\vec{n})\}\ \mathsf{rand}\ \kappa\ N\ \{x.\ x = n * \kappa \hookrightarrow (N,\vec{n})\}}$$

We emphasize that presampling tapes are only used for verification purposes and they do not change the behavior of programs (recall that a labelled rand with an empty tape behaves the same as an unlabelled one). To relate a program annotated with tapes and one without, we provide the two rules HT-COUPLE-RAND-LBL and PUPD-COUPLE-LBL-RAND to couple a pair of labelled and unlabelled rand operations. The rule HT-COUPLE-RAND-LBL couples an unlabelled rand on the left and a labelled one on the right with the empty tape resource on the specification side provided. On

the other hand, the rule PUPD-COUPLE-LBL-RAND couples a presampling action of a tape resource on the implementation side and a rand construct on the right; after which, applying HT-RAND-TAPE later reads out the presampled value in the tape resource.

HT-COUPLE-RAND-LBL

$$\frac{f \text{ bijection} \qquad \forall n \le N.\, \{j \Rightarrow K[f(n)] * \kappa \hookrightarrow_{\mathrm{s}} (N, \epsilon)\}\ n\ \{\Phi\}}{\{j \Rightarrow K[\,\mathsf{rand}\,\kappa\ N] * \kappa \hookrightarrow_{\mathrm{s}} (N, \epsilon)\}\ \mathsf{rand}\ N\ \{\Phi\}}$$

PUPD-COUPLE-LBL-RAND

$$\frac{f \text{ bijection} \quad j \Rightarrow K[\,\mathsf{rand}\,N] \quad \kappa \hookrightarrow (N, \vec{n})}{\rightsquigarrow \exists n \le N.\, \kappa \hookrightarrow (N, \vec{n} \cdot [n]) * j \Rightarrow K[f(n)]}$$

Returning to our example, we prove the refinement via the following intermediate program batchImpl′ annotated with tapes.

$$\begin{aligned}\mathsf{batchImpl}' \triangleq \lambda\_.\ & \mathsf{let}\ (\kappa, \kappa') = (\mathsf{tape}\,N, \mathsf{tape}\,M)\ \mathsf{in} \\ & \mathsf{let}\ (x, y) = (\mathsf{rand}\,\kappa\ N \mid\mid\mid \mathsf{rand}\,\kappa'\ M)\ \mathsf{in} \\ & x \cdot (M+1) + y\end{aligned}$$

We first show that batchImpl refines batchImpl′ by applying HT-COUPLE-RAND-LBL to couple the unlabelled rands on the left with labelled ones on the right; by transitivity of contextual refinement, it then suffices to prove batchImpl′ refines batchSpec. After allocating the tapes in batchImpl′, we then couple two presampling operations on the left with an unlabelled rand on the right with the following rule PUPD-COUPLE-TWO-RANDS-RAND, choosing $f(n, m) \triangleq n \cdot (M+1) + m$. Subsequently, each thread deterministically reads from their corresponding tape resource with HT-RAND-TAPE.

PUPD-COUPLE-TWO-RANDS-RAND

$$\frac{j \Rightarrow K[\,\mathsf{rand}((N+1)\cdot(M+1)-1)] \qquad \kappa \hookrightarrow (N, \vec{n}) \qquad \kappa' \hookrightarrow (M, \vec{m}) \qquad f \text{ bijection}}{\rightsquigarrow \exists n \le N, m \le M.\, j \Rightarrow K[f(n,m)] * \kappa \hookrightarrow (N, \vec{n} \cdot [n]) * \kappa' \hookrightarrow (M, \vec{m} \cdot [m])}$$

Now that we proved the left-to-right refinement, one might ask, can we adopt a similar proof structure with tape presampling for the right-to-left refinement? Unfortunately the answer is *no*! It turns out that, unlike in previous logics supporting tapes [18, 19], it is *unsound* to perform any presampling on the right-hand side program in Foxtrot, and we provide a proof of this in Appendix B. That said, this is not a problem. To see why, consider symbolically stepping through batchImpl now residing on the right-hand side of our refinement. By applying PUPD-PAR-COMP, we then create two thread resources via the parallel composition operator. We now have access to both rand operations as specification resources at the same time, and we can couple the single rand operation in batchSpec with the two rand operations residing in two *separate* threads on the right with the following rule HT-COUPLE-RAND-TWO-RANDS.

HT-COUPLE-RAND-TWO-RANDS

$$\frac{f \text{ bijection} \qquad \forall n \le N, m \le M.\, \{j \Rightarrow K[n] * j' \Rightarrow K'[m]\}\ f(n,m)\ \{\Phi\}}{\{j \Rightarrow K[\,\mathsf{rand}\,N] * j' \Rightarrow K'[\,\mathsf{rand}\,M]\}\ \mathsf{rand}((N+1)\cdot(M+1)-1)\ \{\Phi\}}$$

Let us take a step back and summarize the key ideas of this example. The main challenge of this batch sampling example is that of relating two rand operations with one. Because there is always one left-hand side expression in a Hoare triple, in order for us to reason about multiple rand operations in different threads on the left-hand side program, we utilize presampling tapes and presampling coupling rules to perform the coupling in advance before the actual samplings on the left occur. When the multiple rand operations occur in different threads on the right, we can directly make use of the multiple thread resources produced by the concurrent nature of the program to perform the coupling. We summarize these ideas with the following slogan.

Slogan 2: Tape the left; parallelize the right.

Rejection Sampling. We now turn our attention to another difficult-to-verify pattern in probabilistic computation: rejection sampling schemes. Rejection samplers are a class of Las Vegas algorithms; they generate random samples from a target distribution by repeatedly sampling from another distribution and only accepting a sample if it satisfies certain conditions. Though this rejection sampling technique is ubiquitous, it is also challenging to reason about, particularly due to its use of unbounded looping in the implementation.

Consider the rejection sampler program on the left-hand side of the refinement in Figure 4. Given $M < N$, the rejection sampler program is a function that first samples from $\{0, \ldots, N\}$. We then return the value if it is upper bounded by M, otherwise we repeat the process. It should not be surprising that this function returns an integer from $\{0, \ldots, M\}$ uniformly, and hence one would want to show that it is contextually equivalent to one that calls rand M. We emphasize that this result is *novel* and is stronger than those found in previous work [19], as we show that this contextual equivalence holds even in the concurrent setting.

$$\begin{array}{l}\mathsf{rec}\ f\ \_ = \mathsf{let}\ x = \mathsf{rand}\ N\ \mathsf{in} \\ \mathsf{if}\ x \le M\ \mathsf{then}\ x\ \mathsf{else}\ f\ ()\end{array} \qquad \simeq_{\mathrm{ctx}} \qquad \lambda\ \_.\ \mathsf{rand}\ M$$

Fig. 4. Rejection sampler example.

Here, we sadly cannot use the trick presented in **Slogan 2**; instead of trying to relate two rands with one, we are trying to relate a pair of rands in a somewhat unnatural manner. If the rejection sampler samples a value $\le M$, we want that value to be coupled with rand M. Otherwise if we sampled a value $> M$, we do not want to step rand M at all. In other words, we only want to perform a coupling with the last sampled value in the rejection sampler that gets accepted with rand M.

Foxtrot solves this issue by supporting *fragmented couplings*, a form of couplings first introduced in Approxis [19][4]. To see fragmented couplings in action, let us first consider the left-to-right direction of the refinement. After stepping through the program where we reach rand N on the left and rand M on the right, we apply HT-COUPLE-FRAGMENTED to couple the rand expressions in a fragmented way, taking f as the identity function.

HT-COUPLE-FRAGMENTED

$$\frac{M \le N \qquad f \text{ injective} \qquad \forall n \le N. \left\{ \begin{array}{l} (\exists m \le M.\, f(m) = n * j \Rightarrow K[m]) \vee \\ (\neg(\exists m \le M.\, f(m) = n) * j \Rightarrow K[\,\mathsf{rand}\ M]) \end{array} \right\} n\ \{\Phi\}}{\{j \Rightarrow K[\,\mathsf{rand}\ M]\}\ \mathsf{rand}\ N\ \{\Phi\}}$$

Intuitively, HT-COUPLE-FRAGMENTED performs a case split on the value of rand N. If it is accepted, then we also take a step for rand M on the right, and their values are related by some injective function f (the identity function for this example). Otherwise, we do not take a step on the right. From here, we continue stepping through the proof after a case split. If we sample an accepted value, we can establish the postcondition directly. In the other case, we step the left-hand side program and return to the start of the loop, at which point, we can apply the hypothesis of Löb induction, a standard rule for reasoning about recursive functions in Iris logics [21].

Again, one might ask, can we do the same for the other right-to-left direction of the refinement? Alas, reality is often disappointing and the answer is *no*. This is due to two reasons. The first reason is more technical; we can perform a fragmentation step on the right-hand side program with HT-COUPLE-FRAGMENTED where we can choose to step the program or not, but we cannot

[4]The presentation of the fragmented coupling rules in Foxtrot is however slightly different from those in Approxis. This is because the Approxis fragmented coupling rules are expressed with presampling on the right-hand side program, which is unsound in Foxtrot as shown in Appendix B.

do something similar for the left program due to how the Foxtrot model is defined. Secondly, we cannot apply Löb induction for this direction; in the case that the right-hand side program samples a rejected program, we can only proceed by stepping the right-hand side program back to the start of the recursive function with the left-hand side program kept unchanged. We hence cannot eliminate the later modality in the hypothesis since we do not take any steps on the left program.

To overcome these two limitations, we utilize presampling tapes which we saw previously and *error credits*, a separation logic resource first introduced in Eris [4]. Error credits are written with the $↯(\varepsilon)$ assertion, where ε is a non-negative real. Intuitively, ownership of $↯(\varepsilon)$ error credits means we are attempting to prove a goal allowing some error of at most ε probability. In Foxtrot, error credits are mainly used for a proof technique called *induction by error amplification*, specifically in examples where it is easier to prove that the two programs are equivalent by showing that the distance between them is bounded by errors ε for all $\varepsilon > 0$, like this rejection sampler.

We now review the basic rules of error credits. The rule ERR-SPLIT states that we can split error credits by addition. We can derive False with $↯(1)$ error credits, which holds because any proposition, including False, holds with probability at least 0 (ERR-1). We can generate some arbitrarily-small positive error credits from thin air with PUPD-ERR. Finally, the error amplification rule IND-ERR-AMP allows us to prove any proposition if we are able to amplify some positive error credit by an amplification factor $k > 1$.

$$\text{ERR-SPLIT}\ \frac{↯(\varepsilon_1 + \varepsilon_2)}{↯(\varepsilon_1) \ast ↯(\varepsilon_2)} \qquad \text{ERR-1}\ \frac{↯(1)}{\text{False}} \qquad \text{PUPD-ERR}\ \frac{}{⇝ (\exists \varepsilon.\, 0 < \varepsilon \ast ↯(\varepsilon))} \qquad \text{IND-ERR-AMP}\ \frac{\varepsilon > 0 \qquad k > 1 \qquad ↯(\varepsilon) \qquad (↯(k \cdot \varepsilon) \mathrel{-\!\!\ast} P) \mathrel{-\!\!\ast} ↯(\varepsilon) \mathrel{-\!\!\ast} P}{P}$$

The right-to-left direction of the refinement is established via an intermediate program, similar to the right-to-left refinement of the batch sampling example. In particular our intermediate program is one that matches the ideal rand M program but is modified to use tapes, i.e. $\lambda\ \_.$ rand (tape M) M. It is easy to show that the original rand M program refines this intermediate program with HT-COUPLE-RAND-LBL. Now, for the refinement between the intermediate program and the rejection sampler program, we first apply PUPD-ERR to generate some arbitrarily-small error credit. Then we apply IND-ERR-AMP, choosing $k \triangleq \frac{N+1}{N-M} > 1$. Subsequently we apply PUPD-COUPLE-FRAGMENTED, a rule similar to HT-COUPLE-FRAGMENTED except that it couples a tape presampling operation on the left and a rand N on the right, and it amplifies some error credit by the amplification factor k in the case that we sample a rejected value on the right (we take f to be the identity function).

$$\text{PUPD-COUPLE-FRAGMENTED}\ \frac{M < N \qquad j \Rightarrow K[\,\text{rand}\, N] \qquad \kappa \hookrightarrow (M, \vec{n}) \qquad ↯(\varepsilon) \qquad f \text{ injective}}{⇝ \exists n \le N.\, j \Rightarrow K[n] \ast \left(\begin{array}{l} (\exists m \le M.\, f(m) = n \ast \kappa \hookrightarrow (M, \vec{n} +\!\!+ [n])) \vee \\ \neg(\exists m \le M.\, f(m) = n) \ast \kappa \hookrightarrow (M, \vec{n}) \ast ↯\left(\frac{N+1}{N-M} \cdot \varepsilon\right) \end{array} \right)}$$

The proof then proceeds in a straightforward way after a case split on the result of rand N. If it is accepted, we directly read out the sampled value in the tape on the left-hand side with HT-RAND-TAPE. Otherwise, we step the right-hand program back to the start of the recursive function, and apply the induction hypothesis of IND-ERR-AMP, where we returned to where we started while successfully amplifying the error by k.

To summarize, for programs exhibiting a rejection sampling pattern, we use fragmented couplings to couple the rejection sampling with the simplified ideal one and apply induction by error amplification to show approximate equivalence of the programs for any positive error ε. This brings

us to our third slogan below that captures the key ideas of proving rejection samplers in Foxtrot.

Slogan 3: Fragmented couplings and Error Amplification for Rejections, aka. FEAR.

4 Case Studies

In this section, we present more complex case studies that demonstrate the use of the advanced features of Foxtrot featured in §3 and highlight many subtleties in reasoning about concurrent probabilistic programs. Results in this section are out of scope of previous techniques since they involve a combination of concurrency, probability, and higher-order functions.

4.1 Adversarial von Neumann Coin

John von Neumann [40] showed that a biased coin can be used to simulate the result of flipping a fair coin. This is done by repeatedly tossing the coin twice until the results of a pair of tosses are different, at which point we return the result of the first toss. Zilberstein et al. [43] showed that this result is also true if the bias in stored in shared memory and can be modified as long as it satisfies some invariant, and only fetched right before a pair of tosses. Informally, this means that the program still behaves like a fair coin even when composed with *any* other terminating threads. We formally show that this claim holds by showing contextual equivalence between the following von Neumann coin program and a simple program that samples from rand 1 (see Figure 5).

$$\lambda\, \mathcal{A}.\ \mathsf{let}\ l = \mathsf{ref}\ 0\ \mathsf{in}\ \mathsf{fork}(\mathcal{A}\ l);$$
$$\left(\begin{array}{l} \mathsf{rec}\ f\ \_ = \mathsf{let}\ b = \min\ (!\, l)\ N\ \mathsf{in} \\ \mathsf{let}\ x = \mathsf{rand}(N+1) \le b\ \mathsf{in} \\ \mathsf{let}\ y = \mathsf{rand}(N+1) \le b\ \mathsf{in} \\ \mathsf{if}\ x = y\ \mathsf{then}\ f\ ()\ \mathsf{else}\ x \end{array}\right) \quad \simeq_{\mathrm{ctx}} \quad \lambda\, \mathcal{A}\ \_.\ \mathsf{flip}$$

Fig. 5. Adversarial von Neumann coin example.

Both programs in Figure 5 first take in an adversary argument $\mathcal{A}$ of type ref nat $\to$ unit and returns a closure. Specifically for the adversarial coin program, it creates a reference l containing 0 and forks a thread that calls $\mathcal{A}$ with it. This process allows $\mathcal{A}$ to modify the reference to any natural number in any way, which might involve forking more threads or modifying the reference according to some probabilistic result. The closure returned by the adversarial coin program is a recursive function that sets b to be the minimum of N and the value read from l. This represents setting the bias of the coin to be $(b+1)/(N+1)$, and we flip the coin twice, and return the first result if they differ. If they are the same, we call the closure again.

Most of this proof follows similarly as that of the rejection sampler example seen before (**Slogan 3** in §3.3) albeit a more complicated fragmented coupling rule. In this example, we also have to reason about unknown adversarial code $\mathcal{A}$, which we do so with the *unary* logical relations introduced in §3.1. When proving the left-to-right refinement, we apply Lemma 3.4 to symbolically execute $\mathcal{A}\ l$ in the forked thread on the left. Note that we *cannot* apply the regular binary logical relations to step $\mathcal{A}\ l$ since we cannot couple the adversarial code with anything on the right. The right-to-left direction is even simpler; after forking $\mathcal{A}\ l$, the angelic nondeterminism of the right-hand side program enables us to directly discard the newly-forked thread resource and focus only on the expression in the main thread.

4.2 Sodium

Sodium [12] is a secure cross-platform cross-language open-source cryptography software library. We show the correctness of the randombytes_uniform function in the library for generating uniform

samples (see Figure 6). Our result is particularly strong because it proves that the function behaves as intended even in contexts that might involve *concurrency*, thus showing that the function is *thread-safe* even though its implementation uses a rejection sampling technique and that the random sampling is not atomic (though it appears logically atomic to an outside observer).

The Sodium function first takes in an argument N. We modify our programs slightly by adding in an additional check that N is smaller than $\mathsf{MAX} \triangleq 2^{32}$, which is not needed in the original C implementation since the argument is an unsigned integer of 32 bits. We then perform a simple check on whether N is smaller than 2 and we simply return 0 if that is the case. Otherwise, we take min to be MAX mod N and we repeatedly sample a random number of 32 bits until it is larger or equal to min[5], and we eventually return the remainder of the number divided by N. We show that this somewhat convoluted implementation of a rejection sampler is contextually equivalent to a function that samples from $\{0, \ldots, N-1\}$ directly.

$$\begin{array}{l}\lambda\, N.\ \mathsf{if}\ \mathsf{MAX} \le N\ \mathsf{then}\ 0\\ \quad \mathsf{else\ if}\ N < 2\ \mathsf{then}\ 0\\ \quad \mathsf{else\ let}\ \mathsf{min} = \mathsf{MAX}\ \mathsf{mod}\ N\ \mathsf{in}\\ \quad \mathsf{let}\ r = \mathsf{ref}\ \ 0\ \mathsf{in}\\ \quad \left(\begin{array}{l}\mathsf{rec}\, f\ \_ = r \leftarrow \mathsf{rand}(\mathsf{MAX}-1);\\ \ \mathsf{if}\ !r < \mathsf{min}\ \mathsf{then}\ f\ ()\\ \ \mathsf{else}\ (!r\ \mathsf{mod}\ N)\end{array}\right)()\end{array} \simeq_{\mathrm{ctx}} \begin{array}{l}\lambda\, N.\ \mathsf{if}\ (\mathsf{MAX} \le N || N = 0)\\ \quad \mathsf{then}\ 0\\ \quad \mathsf{else}\ \ \mathsf{rand}(N-1)\end{array}$$

Fig. 6. Sodium randombytes_uniform implementation.

The main bulk of the proof can be split into two stages and it follows very similarly to the arguments we have seen in the batch-sampling and rejection sampler example in §3.3. In the first stage, we show that the implementation of the Sodium function is contextually equivalent to one that samples an integer n from rand $(\mathsf{MAX} - (\mathsf{MAX}\ \mathsf{mod}\ N) - 1)$ and returns n mod N. This is done by applying the fragmented coupling rules captured in **Slogan 3** for reasoning about rejection sampling. Then, in the second stage, we show that this intermediate program p_1 is contextually equivalent to the idealized function that samples from $\mathsf{rand}(N-1)$ directly. This equivalence is established via another intermediate program p_2 that concurrently samples from $\mathsf{rand}(N-1)$ and $\mathsf{rand}(\mathsf{MAX}/N - 1)$ with two threads, but only returns the result of the first thread. For the equivalence between p_1 and p_2, we apply the advanced coupling rules captured in **Slogan 2** to couple the results of two threads with one. Finally, we show that p_2 is contextually equivalent to the idealized program directly with the regular coupling rule HT-COUPLE.

4.3 Other Case Studies

For reasons of space, other case studies demonstrating the flexibility of our approach can be found in the Appendix. For example, similar to a result in Aguirre and Birkedal [3], we show that there exists programs which are contextually equivalent to one another even though there are no optimal schedulers that witness the equivalence (Appendix C). We also present a concurrent one-time pad example that is beyond the scope of all previous techniques, where presampling tapes are utilized to linearize concurrent random samplings (Appendix D). Finally, we study the equations of the algebraic theory induced by the contextual equivalence relation in Foxtrot (Appendix E).

[5]In the actual implementation, this sampling of a random number of 32 bits is done by reading from the **/dev/random** file, which we model directly with a rand.

5 Semantic Model and Soundness

In this section, we describe the semantic model of Foxtrot, defined in the Iris base logic [21], and the key ideas of its soundness proof.

5.1 Full-information Schedulers

As described in §2.3, since ConcRandML is concurrent, we resolve the non-determinism of scheduling the threads through a (stateful probabilistic) scheduler. However, the definition of a scheduler in §2.3 is not easy to work with directly when constructing the model of Foxtrot. Here, we introduce an alternative scheduler called a *full-information schedule* that is only used internally for defining the logic, which we simply write as FIsch(es). We write FIsch to denote the type of FIsches.

Basic Definition of FIsches. From a high-level perspective, FIsches are different from normal schedulers in three aspects. Firstly, the internal state of FIsches are fixed to be $\mathsf{FISchSt} \triangleq \mathit{List}(\mathit{Cfg}' \times \mathbb{N})$, where $\mathit{Cfg}' \triangleq \mathit{List}(\mathit{Expr}) \times \mathit{Heap}$. We implicitly coerce configurations ρ of type Cfg to that of type Cfg' by removing the tape map (Tapes)[6]. Given two FIsch states $\Xi, \Xi' \in \mathsf{FISchSt}$, we write $\Xi \preceq \Xi'$ if Ξ is a prefix of Ξ'.

Secondly, the type of FIsches is different from a normal scheduler. Specifically, FIsches are defined by a transition function Υ of the type $(\mathsf{FISchSt} \times \mathit{Cfg}) \to \mathsf{option}(\mathcal{D}(\mathbb{N}))$.

Lastly, all FIsches Υ satisfy a *consistency* condition, where for all FIsch states Ξ and configurations ρ, if $\Upsilon(\Xi, \rho) = \mathsf{None}$, then for all other configurations ρ' we have $\Upsilon(\Xi, \rho') = \mathsf{None}$.

Semantics of FIsches. We modify the operational semantics of ConcRandML with respect to FIsches, such that the thread-stepping function FItpStep of FIsches is more "eager", i.e. it attempts to step the corresponding thread regardless of whether the configuration is final or not:

$$\mathsf{FItpStep}(\vec{e}, \sigma)(j) \triangleq \begin{cases} \mathsf{ret}(\vec{e}, \sigma) & \text{if } e_j \in \mathit{Val} \text{ or } j \geq |\vec{e}|\,, \\ \mathsf{step}(e_j, \sigma) \gg\!= \lambda\, (e_j', \sigma', \vec{e}').\ \mathsf{ret}(\vec{e}[j \mapsto e_j'] +\!\!+ \vec{e}', \sigma') & \text{otherwise.} \end{cases}$$

Given a configuration ρ, a FIsch Υ, and a FIsch state Ξ, we can now define the single FIsch-step reduction function $\mathsf{FIschStep}_\Upsilon(\Xi, \rho) \in \mathcal{D}(\mathit{SchedSt} \times \mathit{Cfg})$ as follows:

$$\mathsf{FIschStep}_\Upsilon(\Xi, \rho) \triangleq \begin{cases} \mu \gg\!= \lambda\, j.\ \mathsf{FItpStep}(\rho, j) \gg\!= \lambda\, \rho'.\ \mathsf{ret}(\Xi \cdot [(\rho, j)], \rho') & \Upsilon(\Xi, \rho) = \mathsf{Some}\ \mu \\ \mathsf{ret}(\Xi, \rho) & \text{otherwise} \end{cases}$$

Here, in addition to stepping the configuration according to the FIsch Υ, FIschStep also updates the FIsch's internal state by extending it with the configuration ρ and the thread j it attempts to step. Intuitively, this captures the idea that FIsches are keeping track of all the information it has access to since the beginning of the execution, hence the term "full-information".

The n-step program execution of a FIsch Υ returns both its internal state and the configuration if the FIsch reaches a None state:

$$\mathsf{FIexec}_{\Upsilon,n}(\Xi, \rho) = \begin{cases} \mathsf{ret}(\Xi, \rho) & \Upsilon(\Xi, \rho) = \mathsf{None} \\ \mathbf{0} & n = 0 \\ \mathsf{FIschStep}_\Upsilon(\Xi, \rho) \gg\!= \mathsf{FIexec}_{\Upsilon,n-1} & \text{otherwise.} \end{cases}$$

In contrast to the n-step program execution of a normal scheduler, $\mathsf{FIexec}_{\Upsilon,n}$ returns a distribution on the FIsch state and configuration, instead of a distribution of values. Intuitively, $\mathsf{FIexec}_{\Upsilon,n}$ simulates the n steps of the execution of a FIsch Υ until it returns a None. The full program execution of a FIsch is taken as the limit of its n-step program execution, i.e. $\mathsf{FIexec}_\Upsilon(\Xi, \rho) \triangleq \lim_{n\to\infty} \mathsf{FIexec}_{\Upsilon,n}(\Xi, \rho)$.

[6] We remove the tape map to ensure FIsches cannot schedule threads based on the content of the tapes.

Properties of FIsches. As mentioned, FIsches are only used internally to define the model of Foxtrot. We relate FIsches and our original definition of schedulers via the following lemma:

Lemma 5.1. *Given a FIsch Υ, there exists a (regular) scheduler ζ such that for all FIsch states Ξ and values v, we have* $(\mathsf{Flexec}_\Upsilon(\Xi,\rho) \gg\!= g)(v) \leq \mathsf{exec}_\zeta(\Xi,\rho)(v)$ *where*

$$g \triangleq \lambda\,(\Xi',\rho').\ \begin{cases}\mathrm{ret}\, v & v \in \mathit{Val} \wedge \rho'.1 = v \cdot \vec{e}\\ \mathbf{0} & \mathit{otherwise}\end{cases}$$

We now consider various ways to construct FIsches, which are used in proving the soundness of the model of Foxtrot and its proof rules.

The FIschLift function takes a FIsch Υ and FIsch state Ξ as input, and returns a FIsch that behaves the same as Υ but as if all states are additionally prepended with Ξ in front.

Lemma 5.2. *There exists a function FIschLift such that given FIsch Υ and FIsch state Ξ, FIschLift(Ξ,Υ) is a FIsch where for all scheduler state Ξ' and configurations ρ, we have* $\mathsf{Flexec}_{\mathit{FIschLift}(\Xi,\Upsilon)}(\Xi\cdot\Xi',\rho) = \Upsilon(\Xi',\rho) \gg\!= \lambda\,(\Xi'',\rho').\,(\Xi\cdot\Xi'',\rho')$.

We also define the FIschApp function allowing us to chain a FIsch with a family of FIsches. Intuitively, FIschApp(Υ, f) is a FIsch that first acts like Υ until we reach a state Ξ_2 for which Υ always returns None, at which point, we continue to behave as the Fisch $f(\Xi_2)$.

Lemma 5.3. *There exists a function FIschApp such that given FIsch Υ and a function f of type* $\mathsf{FISchSt} \to \mathsf{FIsch}$, *FIschApp$(\Upsilon, f)$ is a FIsch where for all ρ, x, we have* $(\mathsf{Flexec}_\Upsilon(\epsilon,\rho) \gg\!= g)(x) \leq \mathsf{Flexec}_{\mathit{FIschApp}(\Upsilon,f)}(\epsilon,\rho)(x)$ *where*

$$g(\Xi_1,\rho_1) \triangleq \begin{cases}\mathsf{Flexec}_{\mathit{FIschLift}(\Xi_2, f(\Xi_2))}(\Xi_1,\rho_1) & \mathit{exists}\ \Xi_2 \leq \Xi_1\ \mathit{minimal\ s.t.}\ \forall\rho'.\Upsilon(\Xi_2,\rho') = \mathsf{None}\\ \mathbf{0} & \mathit{otherwise}\end{cases}$$

5.2 Semantic Model

Weakest Precondition. We follow the standard trick of expressing Hoare triples in terms of a weakest precondition [21], i.e. $\{P\}\ e\ \{Q\} \triangleq P \mathrel{-\!\!*} \mathsf{wp}\ e\ \{Q\}$. The definition of the weakest precondition (shown below) follows similar structure as previous logics [19, 26] where it is defined as a guarded fixed point (all recursive occurrences of the weakest precondition appears under the later modality $\triangleright$). We emphasize that novelty of the Foxtrot weakest precondition lies in the significant differences of how the spec-coupling (sstep) and program-coupling (pstep) preconditions are defined, which we detail later.

$$\begin{aligned}\mathsf{wp}\ e_1\ \{\Phi\} \triangleq \forall\sigma_1,\rho_1,\varepsilon_1.\ & S\ \sigma_1\ \rho_1\ \varepsilon_1 \mathrel{-\!\!*} {}_\top\!\Vdash\!\!\!\Rightarrow_\emptyset \mathsf{sstep}\ \sigma_1\ \rho_1\ \varepsilon_1\ \{\sigma_2,\rho_2,\varepsilon_2.\\ & \big(e_1 \in \mathit{Val} * {}_\emptyset\!\Vdash\!\!\!\Rightarrow_\top S\ \sigma_2\ \rho_2\ \varepsilon_2 * \Phi\ e_1\big) \vee\\ & \big(e_1 \notin \mathit{Val} * \mathsf{pstep}\ (e_1,\sigma_2)\ \rho_2\ \varepsilon_2\ \{e_2,\sigma_3,l,\rho_3,\varepsilon_3.\\ & \quad \triangleright\, \mathsf{sstep}\ \sigma_3\ \rho_3\ \varepsilon_3\ \{\sigma_4,\rho_4,\varepsilon_4.\ {}_\emptyset\!\Vdash\!\!\!\Rightarrow_\top S\ \sigma_4\ \rho_4\ \varepsilon_4 * \mathsf{wp}\ e_2\ \{\Phi\} * \mathop{\Large *}_{e'\in l} \mathsf{wp}\ e'\ \{\mathsf{True}\}\}\}\big)\}\end{aligned}$$

To prove a weakest precondition, we initially assume the ownership of the *state interpretation* $S\ \sigma_1\ \rho_1\ \varepsilon_1$, which interprets the physical state of the implementation program, the specification program, and the approximate error as resources in the standard way [19, 21, 26]. It hence gives meaning to various resources e.g. the points-to connective $\ell \mapsto v$, the specification thread connective $j \Rightarrow e$, and the error credits $\$(\varepsilon)$. We then get access to the resources in all invariants through an update modality ${}_\top\!\Vdash\!\!\!\Rightarrow_\emptyset$ and we need to prove a *spec-coupling* precondition $\mathsf{sstep}\ \sigma_1\ \rho_1\ \varepsilon_1\{\dots\}$. We explain its definition later in Figure 7; for now it suffices to interpret it as a precondition for allowing the right-hand side program to progress.

$$\textsc{spec-step-err-1}\ \frac{1 \le \varepsilon}{\mathsf{sstep}\ \sigma\ \rho\ \varepsilon\ \{\Phi\}} \qquad \textsc{spec-step-ret}\ \frac{\Phi(\sigma,\rho,\varepsilon)}{\mathsf{sstep}\ \sigma\ \rho\ \varepsilon\ \{\Phi\}} \qquad \textsc{spec-step-continuous}\ \frac{\forall \varepsilon'.\ \varepsilon < \varepsilon' \twoheadrightarrow \mathsf{sstep}\ \sigma\ \rho\ \varepsilon'\ \{\Phi\}}{\mathsf{sstep}\ \sigma\ \rho\ \varepsilon\ \{\Phi\}}$$

$$\textsc{spec-step-exp}\ \frac{\mathsf{schErasable}(\mu,\sigma_1) \qquad \exists r.\ \forall x.\ \mathcal{F}_2(x) \le r \qquad (\varepsilon' + \mathbb{E}_{\mathsf{Flexec}_\Upsilon(\epsilon,\rho_1)}[\mathcal{F}]) \le \varepsilon \qquad \mu \lesssim_{\varepsilon'} \mathsf{Flexec}_\Upsilon(\epsilon,\rho_1) : R \qquad \mathsf{stateinj}(R) \qquad \forall \sigma_2, (\Xi,\rho_2).\ R\ \sigma_2\ (\Xi,\rho_2) \twoheadrightarrow \mathsf{sstep}\ \sigma_2\ \rho_2\ (\mathcal{F}(\Xi,\rho_2))\ \{\Phi\}}{\mathsf{sstep}\ \sigma_1\ \rho_1\ \varepsilon\ \{\Phi\}}$$

Fig. 7. Inductive Definition of the Spec Step Precondition $\mathsf{sstep}\ \sigma\ \rho\ \varepsilon\ \{\Phi\}$.

We subsequently perform a case split on whether e_1 is a value. If it is, we do a view shift ${}_{\emptyset}\!\models\!\!\Rightarrow_{\top}$ where we re-establish all invariants, return the updated state interpretation and prove that e_1 satisfies the postcondition Φ. Otherwise, we prove a program step precondition $\mathsf{pstep}\ (e_1,\sigma_2)\ \rho_2\ \varepsilon_2\{\dots\}$. As before, we explain the details of this precondition later, but for now one can think of it as a precondition similar to that of sstep, but it allows us to also take an actual step on the configuration (e_1,σ_2) to obtain resulting expression e_2, state σ_3, a list of forked expressions l, and resulting error budget ε_3. We then prove another sstep precondition, which can be ignored here and is only needed for certain invariant opening properties not presented in this paper. We finally re-establish all invariants through the view shift ${}_{\emptyset}\!\models\!\!\Rightarrow_{\top}$ and return the newly updated state interpretation, prove that $\mathsf{wp}\ e_2\ \{\Phi\}$ holds, and show that all forked expressions are safe to execute, i.e. $\mathsf{wp}\ e'\ \{\mathsf{True}\}$ for every expression e' in the list l.

Coupling Preconditions. We now turn our attention to the spec-coupling and program-coupling preconditions. We first present two auxiliary definitions.

Firstly, we say a relation $R \subseteq A \times (B \times C)$ is state injective, written as $\mathsf{stateinj}(R)$, if it is functional in B, i.e. for all a, a', b, c, c', if $R\ a\ (b,c)$ and $R\ a'\ (b,c')$, then $a = a'$ and $c = c'$.

Secondly, we define the notion of a scheduler erasable state update, which intuitively describes actions on the state (mostly on the tape heap) that do not alter the operational semantics of the expressions; examples of scheduler erasable state updates include tape presamplings actions.

Definition 5.4. *A distribution on states μ is a* scheduler erasable state update *of $\sigma \in$ State, written as* $\mathsf{schErasable}(\mu,\sigma)$*, if for all schedulers ζ, scheduler states Ξ, thread pools $\vec{e}$, and integers n, we have*

$$(\mu \gg\!= (\lambda\, \sigma'.\ \mathsf{pexec}_{\zeta,n}(\Xi,(\vec{e},\sigma')))).\mathsf{tp} \ = \ (\mathsf{pexec}_{\zeta,n}(\Xi,(\vec{e},\sigma))).\mathsf{tp}$$

where the function $-.\mathsf{tp}$ projects out the thread pool component from a distribution on configurations.

The spec-coupling precondition sstep is defined inductively by four inference rules shown in Figure 7. If the error budget ε is lower-bounded by 1, the precondition holds trivially as all sub-distributions have mass smaller or equal to 1 (SPEC-STEP-ERR-1). If the postcondition holds for the input parameters, the precondition also holds trivially (SPEC-STEP-RET). If the precondition holds for any error budget ε' larger than ε, then the precondition also holds for ε (SPEC-STEP-CONTINUOUS). This rule enables us to prove PUPD-ERR and create error credits from thin air.

We now focus on SPEC-STEP-EXP, which is the most important inference rule in the sstep precondition. Intuitively, this rule establishes an ε'-approximate coupling between a scheduler erasable state update μ, e.g. a tape presampling action on the left, with some execution steps of the right-hand side specification program under some state injective relation R. Here, the execution of the specification program is captured by the limit execution of a FIsch Υ with the empty list ϵ as the starting state. In addition, the rule allows us to distribute the remaining error according to some function $\mathcal{F}$ as long as it satisfies an expectation-preserving inequality.

The program-coupling precondition pstep is defined by the single inference rule PROG-STEP-EXP which is of a similar structure as that of SPEC-STEP-EXP. The main difference here is that this rule

couples a single execution step on the left-hand side program with some execution steps on the right-hand side program according to some FIsch Υ. (The proposition $\mathsf{red}(e, \sigma)$ means that the configuration (e, σ) is reducible for one execution step.)

PROG-STEP-EXP

$$\frac{\begin{array}{ll}\exists r.\, \forall x.\, \mathcal{F}(x) \le r & (\varepsilon' + \mathbb{E}_{\mathsf{Flexec}_\Upsilon(\epsilon,\rho_1)}[\mathcal{F}]) \le \varepsilon \qquad \mathsf{step}(e_1,\sigma_1) \lesssim_{\varepsilon'} \mathsf{Flexec}_\Upsilon(\epsilon,\rho_1) : R \\ \mathsf{red}(e_1,\sigma_1) \qquad \mathsf{stateinj}(R) & \forall (e_2,\sigma_2,l), (\Xi,\rho_2).\, R\ (e_2,\sigma_2,l)\ (\Xi,\rho_2) \mathrel{-\!\!*} \Phi(e_2,\sigma_2,l,\rho_2,\mathcal{F}(\Xi,\rho_2))\end{array}}{\mathsf{pstep}\ (e_1,\sigma_1)\ \rho_1\ \varepsilon\ \{\Phi\}}$$

Probabilistic Update Modality. We follow the approach of Coneris [26] in defining the probabilistic update modality. Just like the fancy update modality ${}_{\mathcal{E}_1}\!\Rightarrow_{\mathcal{E}_2}$, the probabilistic update modality is in fact additionally annotated with two sets of invariants, i.e. ${}_{\mathcal{E}_1}\!\Rightarrow\!\!\!\!\!\sim_{\mathcal{E}_2}$. Intuitively, ${}_{\mathcal{E}_1}\!\Rightarrow\!\!\!\!\!\sim_{\mathcal{E}_2} P$ is defined by first assuming ownership of some state interpretation, opening all invariants in $\mathcal{E}_1$, proving a spec-coupling precondition with input parameters σ_1, ρ_1, and ε_1, re-establishing all invariants in the mask $\mathcal{E}_2$, and finally giving back the state interpretation and resource P.

$$ {}_{\mathcal{E}_1}\!\Rightarrow\!\!\!\!\!\sim_{\mathcal{E}_2} P \triangleq \forall \sigma_1, \rho_1, \varepsilon_1.\, S\ \sigma_1\ \rho_1\ \varepsilon_1 \mathrel{-\!\!*} {}_{\mathcal{E}_1}\!\Rightarrow_{\emptyset} \mathsf{sstep}\ \sigma_1\ \rho_1\ \varepsilon_1\ \{\sigma_2, \rho_2, \varepsilon_2.\, {}_{\emptyset}\!\Rightarrow_{\mathcal{E}_2} S\ \sigma_2\ \rho_2\ \varepsilon_2 * P\} $$

5.3 Soundness

As hinted in §1.2, to compose families of FIsches, we rely on an Iris version of the axiom of choice:

Lemma 5.5 (Iris Choice). *Assuming the axiom of choice, we have* $\forall a.\, \exists b.\, P\ a\ b \vdash \exists f.\, \forall a.\, P\ a\ (f(a))$ *within the Iris separation logic.*

Perhaps this result might appear suspicious at first glance. Diaconescu [13] showed that the original axiom of choice implies the law of excluded middle, which is known to not hold in the intuitionistic Iris separation logic [23]. The seeming contradiction is avoided by restricting Lemma 5.5 to only apply when the variables a and b are quantified over meta-level Rocq types. Consequently, the types of a and b cannot utilize the step-indexing and resources of the Iris logic, as would be necessary to replay Diaconescu's proof.

The adequacy theorem (Theorem 3.1) is proven via the following intermediate lemma Lemma 5.6[7]. We then recover Theorem 3.1 by first taking $\vec{e}$ to be the empty list ϵ, applying Lemma 5.1 to eliminate the use of FIsches, and finally taking the limit of n.

Lemma 5.6. *If* $\not\zeta(\varepsilon) * \mathop{\Large\ast}_{(j,e')\in\mathsf{enum}(\vec{e_s})} j \Mapsto e' \vdash \mathsf{wp}\ e\ \{v. \exists v'.\, 0 \Mapsto v' * \phi\ v\ v'\} * \mathop{\Large\ast}_{e'\in\vec{e}} \mathsf{wp}\ e'\ \{\mathsf{True}\}$, *then for all schedulers* ζ, *scheduler states* Ξ, *states* σ, σ', *natural numbers* n, *and error* $\varepsilon' > 0$, *there exists a FIsch* Υ *such that* $\mathsf{exec}_{\zeta,n}(\Xi, (e \cdot \vec{e}, \sigma)) \lesssim_{\varepsilon+\varepsilon'} \mathsf{Flexec}_\Upsilon(\epsilon, (\vec{e_s}, \sigma')) : \psi$ *where*

$$\psi\ v\ (\Xi', \rho') \triangleq \exists v'\ \vec{e}'.\, \rho'.1 = v' \cdot \vec{e}' \wedge \phi\ v\ v'$$

Intuitively, Lemma 5.6 states that for any scheduler ζ on the left and any execution steps n, we can find a FIsch Υ for the right such that we can couple the n-step execution of the left-hand side program with the limit execution of the one on the right. The proof proceeds by induction on n, and during the inductive case, we repeatedly apply Lemma 5.5 and the FIschApp function to combine a family of FIsches into one. (In fact, this procedure has to be done a total of four times in the proof of the inductive case!) We provide a proof sketch of Lemma 5.6 in Appendix F.

[7] We quantify over errors $\varepsilon' > 0$ because sometimes an optimal scheduler cannot be found for the right-hand side program, but we can find one that approximately couples with the left for any positive error, see Appendix C for more details.

6 Related Work

Contextual Equivalences of Higher-Order Programs. There are various prior work in utilizing binary logical relations to prove contextual equivalences of higher-order programs. For example, there is a long line of work in utilizing step-indexed logical relations for establishing refinements of higher-order concurrent languages. In particular, we highlight CaReSL [38], one of the first logics for fine-grained concurrency, which was then mechanized in Iris [25, 36]. The mechanized concurrent separation logic ReLoC [17] extended CaReSL to internalize refinement judgements as first-class logical statements, providing more concise rules for reasoning about invariants and logical atomicity. Similarly, there are also many logical relations specialized for probabilistic languages. Bizjak and Birkedal [7] defined a step-indexed, biorthogonal logical relation for reasoning about contextual equivalences of a higher-order probabilistic language with discrete random variables and state. This was then extended for more complicated language features such as continuous random sampling [10, 41] and countable nondeterminism [3]. The mechanized logic Clutch [18] first introduced presampling tapes for asynchronous couplings, which we also use to implement advanced coupling rules for reasoning about rejection sampling and multiple concurrent samplings in Foxtrot. We also took inspiration from Approxis [19], which used error credits to establish contextual equivalences of rejection samplers by means of approximation.

There are other techniques other than logical relations to verify contextual equivalences, e.g. Ehrhard et al. [14] utilized probabilistic coherence spaces to construct a fully abstract denotational model for PCF programs with a discrete random primitive. Much work has also been done in using bisimulation techniques to establish the contextual equivalences of programs written in the probabilistic λ-calculus, whether it be call-by-value [9], call-by-name [11], or call-by-need [31].

Compared to all the work above, our work in proving contextual equivalences of higher-order (stateful) programs is the *first* to be extended to the concurrent and randomized setting, which we believe is a *significant* contribution as reasoning about this mixture is notoriously hard.

Logics for Probability and Concurrency. There has been previous work on program logics for reasoning about properties (other than contextual refinement) of concurrent probabilistic programs.

McIver et al. [29] first introduced the probabilistic rely-guarantee calculus for reasoning about the quantitative correctness of probabilistic concurrent programs without support for dynamically-allocated local state. Fesefeldt et al. [16] developed concurrent quantitative separation logic for reasoning about quantitative properties of concurrent, randomized, heap-manipulating programs. Polaris [33] is similar to our work in that it is also a mechanized relational logic in Iris, but the refinement is constructed between a concurrent probabilistic program and a simpler monadic representation model, which can then further be analyzed to establish properties of the original program, e.g. bounds on expected values. ExpIris [28] is another mechanized program logic in Iris; its Hoare triples are annotated with a parameter called the potential, which is used to prove bounds on expected costs of concurrent probabilistic programs. The recent Probabilistic Concurrent Outcome Logic [43], a variant of Outcome Logic [42], supports compositional reasoning of the distribution of possible outcomes from the execution of concurrent randomized programs.

Most similar to our logic is that of Coneris [26], a separation logic for proving error bounds of concurrent probabilistic programs. Firstly, the underlying language ConcRandML of Foxtrot is identical to that in Coneris. Moreover, our use of error credits is heavily inspired by that of Coneris [26] (and similar logics such as Eris [4] and Approxis [19]), where the error credits were originally used to prove error bounds of (concurrent) probabilistic programs. We also adopt Coneris' probabilistic update modality and extend it into our relational setting for symbolically executing right-hand side programs in addition to presampling onto tapes.

7 Conclusion

We presented Foxtrot, the first higher-order separation logic for proving contextual refinements of higher-order concurrent probabilistic programs with higher-order local state. In addition, we demonstrated its strengths on a wide range of examples involving complex local state, random sampling, and concurrent behavior, that are previously out of scope.

In the future, we would like to extend Foxtrot to reason about contextual refinement of programs under schedulers with restricted power for proving security guarantees of distributed cryptographic protocols, e.g. schedulers which can only rely on its internal state to decide which thread to step next. We also aim to implement prophecy variables [1, 2, 22] within Foxtrot so to prove more interesting equivalences. Lastly, it would be interesting to consider modifying Foxtrot for proving must-termination contextual refinements [3, 6] of concurrent probabilistic programs.

Acknowledgments

The first author would like to thank Amin Timany and Daniel Gratzer for insightful discussions regarding the axiom of choice.

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

References

[1] M. Abadi and L. Lamport. 1988. The existence of refinement mappings. In *[1988] Proceedings. Third Annual Symposium on Logic in Computer Science*. 165–175. doi:10.1109/LICS.1988.5115

[2] Martín Abadi and Leslie Lamport. 1991. The existence of refinement mappings. *Theoretical Computer Science* 82, 2 (1991), 253–284. doi:10.1016/0304-3975(91)90224-P

[3] Alejandro Aguirre and Lars Birkedal. 2023. Step-Indexed Logical Relations for Countable Nondeterminism and Probabilistic Choice. *Proceedings of the ACM on Programming Languages* 7 (9 Jan. 2023), 33–60. doi:10.1145/3571195 Publisher Copyright: © 2023 Owner/Author..

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

[5] James Aspnes and Eric Ruppert. 2016. Depth of a Random Binary Search Tree with Concurrent Insertions. In *Distributed Computing*, Cyril Gavoille and David Ilcinkas (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 371–384.

[6] Lars Birkedal, Aleš Bizjak, and Jan Schwinghammer. 2013. Step-Indexed Relational Reasoning for Countable Nondeterminism. *Logical Methods in Computer Science* Volume 9, Issue 4 (Oct. 2013). doi:10.2168/lmcs-9(4:4)2013

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

[8] Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. 2013. Coquelicot: A User-Friendly Library of Real Analysis for Coq. (Sept. 2013). https://inria.hal.science/hal-00860648 working paper or preprint.

[9] Raphaëlle Crubillé and Ugo Dal Lago. 2014. On Probabilistic Applicative Bisimulation and Call-by-Value λ-Calculi. In *Programming Languages and Systems*, Zhong Shao (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 209–228.

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

[11] Ugo Dal Lago, Davide Sangiorgi, and Michele Alberti. 2014. On coinductive equivalences for higher-order probabilistic functional programs. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 297–308. doi:10.1145/2535838.2535872

[12] Frank Denis. 2013. The Sodium cryptography library. https://download.libsodium.org/doc/

[13] Radu Diaconescu. 1975. Axiom of Choice and Complementation. *Proc. Amer. Math. Soc.* 51, 1 (1975), 176–178. http://www.jstor.org/stable/2039868

[14] Thomas Ehrhard, Christine Tasson, and Michele Pagani. 2014. Probabilistic coherence spaces are fully abstract for probabilistic PCF. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 309–320. doi:10.1145/2535838.2535865

[15] David Evans, Vladimir Kolesnikov, and Mike Rosulek. 2018. . doi:10.1561/3300000019

[16] Ira Fesefeldt, Joost-Pieter Katoen, and Thomas Noll. 2022. Towards Concurrent Quantitative Separation Logic. In *33rd International Conference on Concurrency Theory (CONCUR 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 243)*, Bartek Klin, Sławomir Lasota, and Anca Muscholl (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 25:1–25:24. doi:10.4230/LIPIcs.CONCUR.2022.25

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

[18] 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 (Jan. 2024), 32 pages. doi:10.1145/3632868

[19] Philipp G. Haselwarter, Kwing Hei Li, Alejandro Aguirre, Simon Oddershede Gregersen, Joseph Tassarotti, and Lars Birkedal. 2025. Approximate Relational Reasoning for Higher-Order Probabilistic Programs. *Proc. ACM Program. Lang.* 9, POPL, Article 41 (Jan. 2025), 31 pages. doi:10.1145/3704877

[20] 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. doi:10.1145/2951913.2951943

[21] 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. doi:10.1017/S0956796818000151

[22] Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. 2019. The future is ours: prophecy variables in separation logic. *Proc. ACM Program. Lang.* 4, POPL, Article 45 (Dec. 2019), 32 pages. doi:10.1145/3371113

[23] 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. doi:10.1145/2676726.2676980

[24] Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017. 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. doi:10.1007/978-3-662-54434-1_26

[25] Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. 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*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 205–217. doi:10.1145/3009837.3009855

[26] Kwing Hei Li, Alejandro Aguirre, Simon Oddershede Gregersen, Philipp G. Haselwarter, Joseph Tassarotti, and Lars Birkedal. 2025. Modular Reasoning about Error Bounds for Concurrent Probabilistic Programs. *Proc. ACM Program. Lang.* 9, ICFP, Article 245 (Aug. 2025), 30 pages. doi:10.1145/3747514

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

[28] Janine Lohse and Deepak Garg. 2024. An Iris for Expected Cost Analysis. arXiv:2406.00884 [cs.PL]

[29] Annabelle McIver, Tahiry Rabehaja, and Georg Struth. 2016. Probabilistic rely-guarantee calculus. *Theoretical Computer Science* 655 (2016), 120–134. doi:10.1016/j.tcs.2016.01.016 Quantitative Aspects of Programming Languages and Systems (2013-14).

[30] Robert Morris. 1978. Counting large numbers of events in small registers. *Commun. ACM* 21, 10 (Oct. 1978), 840–842. doi:10.1145/359619.359627

[31] David Sabel, Manfred Schmidt-Schauß, and Luca Maio. 2022. Contextual Equivalence in a Probabilistic Call-by-Need Lambda-Calculus. In *Proceedings of the 24th International Symposium on Principles and Practice of Declarative Programming* (Tbilisi, Georgia) *(PPDP '22)*. Association for Computing Machinery, New York, NY, USA, Article 4, 15 pages. doi:10.1145/3551357.3551374

[32] Tetsuya Sato. 2016. Approximate Relational Hoare Logic for Continuous Random Samplings. In *The Thirty-second Conference on the Mathematical Foundations of Programming Semantics, MFPS 2016*. doi:10.1016/J.ENTCS.2016.09.043

[33] Joseph Tassarotti and Robert Harper. 2019. A separation logic for concurrent randomized programs. *Proc. ACM Program. Lang.* 3, POPL, Article 64 (Jan. 2019), 30 pages. doi:10.1145/3290377

[34] The Rocq Development Team. 2024. *The Rocq Prover.* doi:10.5281/zenodo.11551307

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

[36] A. Timany. 2018. *Contributions in Programming Languages Theory: Logical Relations and Type Theory.* Ph. D. Dissertation. , Faculty of Engineering Science, Science, Engineering and Technology Group. https://lirias.kuleuven.be/1838165 Jacobs, B. (supervisor), Piessens, F. (cosupervisor).

[37] Amin Timany, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2024. A Logical Approach to Type Soundness. *J. ACM* 71, 6 (2024), 40:1–75.

[38] Aaron J. Turon, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer. 2013. Logical relations for fine-grained concurrency. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13)*. Association for Computing Machinery, New York, NY, USA, 343–356. doi:10.1145/2429069.2429111

[39] Daniele Varacca and Glynn Winskel. 2006. Distributing probability over non-determinism. *Mathematical. Structures in Comp. Sci.* 16, 1 (Feb. 2006), 87–113. doi:10.1017/S0960129505005074

[40] John von Neumann. 1961. *John von Neumann Collected Works.* Vol. 5: Design of Computers, Theory of Automata and Numerical Analysis. Pergamon Press, Oxford, England, Chapter Various Techniques Used in Connection with Random Digits, 768–770. Notes by George E. Forsythe and reproduced from J. Res. Nat. Bus. Stand. Appl. Math. Series, vol. 3, pp. 36–38 (1955)..

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

[42] Noam Zilberstein, Derek Dreyer, and Alexandra Silva. 2023. Outcome Logic: A Unifying Foundation for Correctness and Incorrectness Reasoning. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 93 (April 2023), 29 pages. doi:10.1145/3586045

[43] Noam Zilberstein, Alexandra Silva, and Joseph Tassarotti. 2024. Probabilistic Concurrent Reasoning in Outcome Logic: Independence, Conditioning, and Invariants. arXiv:2411.11662 [cs.LO] https://arxiv.org/abs/2411.11662

A Auxiliary Programs

We introduce two auxiliary programs (Figure 8): nondet and diverge.

$$\begin{array}{l}\text{nondet} \triangleq \lambda\_. \\ \quad \text{let } x = \text{ref } 0 \text{ in} \\ \quad \text{fork}((\text{rec } f\ \_ = x \leftarrow !x + 1; f\ ())()) \\ \quad !x\end{array} \qquad \text{diverge} \triangleq \text{rec } f\ \_ = f\ ()$$

Fig. 8. The nondet and diverge programs.

The program nondet is a function that first creates a reference containing 0. It then spawns a thread that repeatedly increments the reference, and the main thread reads from the reference and returns the value. It is trivial to see that for any natural number n, there exists some scheduling such that nondet () returns n, as captured by the rule HT-NONDET. In fact, if this program is on the right-hand side of the refinement, we can deliberately choose the threads to step in a way to return a particular natural number, as captured by PUPD-NONDET.

The program diverge is simply a recursive program that calls itself. With Löb induction, one can prove HT-DIVERGE, since it never terminates with a value. There is no corresponding rule for diverge () appearing on the right-hand side of the refinement.

HT-NONDET

$$\frac{}{\{\text{True}\}\ \text{nondet}\ ()\ \{v.\exists(n:\mathbb{N}).\ v = n\}}$$

PUPD-NONDET

$$\frac{j \Mapsto K[\text{nondet}\ ()]}{\rightsquigarrow j \Mapsto K[n] \quad (\text{for any natural number } n)}$$

HT-DIVERGE

$$\frac{}{\{\text{True}\}\ \text{diverge}\ ()\ \{\text{False}\}}$$

B Counter example of presampling on the right

We prove that presampling on the right hand side is unsound in Foxtrot. Assume we have the following unsound coupling rule, that allows you to couple a rand on the left and a presampling on a tape on the right that is sound in previous logics supporting tapes [18, 19]:

UNSOUND-HT-COUPLE-RAND-LBL

$$\frac{\forall n \le N.\ \{\kappa \hookrightarrow_s (N, \vec{n} \cdot [n])\}\ n\ \{\Phi\}}{\{\kappa \hookrightarrow_s (N, \vec{n})\}\ \text{rand}\ N\ \{\Phi\}}$$

We now define the following four program:

$$\begin{array}{l}\text{progA} \triangleq () \\ \text{progB} \triangleq \text{if nondet}\ () = \text{rand}\ 1 \text{ then } () \text{ else diverge}\ () \\ \text{progC} \triangleq \text{let } \kappa = \text{tape}\ 1 \text{ in} \\ \qquad \text{if rand}\ \kappa\ 1 = \text{nondet}\ () \text{ then } () \text{ else diverge}\ () \\ \text{progD} \triangleq \text{if rand}\ 1 = 1 \text{ then } () \text{ else diverge}\ ()\end{array}$$

Note that programs in ConcRandML follow a right-to-left evaluation order, so the expression nondet () = rand 1 first performs the random sampling before executing nondet (). We can prove the following three inequalities:

(1) The upper termination probability of progA is upper bounded by that of progB. This is done by forcing nondet () to return the same value produced by rand 1.

(2) The upper termination probability of progB is upper bounded by that of progC. We can prove this with our unsound rule where we perform the coupling between the labelled rand on the left and the presampling action on the tape on the specification side. We also force nondet () to return the value presampled onto the tape. This rest of the proof follows by stepping the programs symbolically.
(3) The upper termination probability of progC is upper bounded by that of progD. This proof is done in a straightforward manner by choosing the right coupling between the two rand operations after executing nondet () on the left.

By transitivity, it follows that the upper termination probability of progA is upper bounded by that of progD. This is a contradiction since progA almost surely terminates but progD does not.

C Example without Optimal Scheduling

Aguirre and Birkedal [3] showed that there are pairs of non-deterministic probabilistic programs that are contextually equivalent with each other even though there does not exist an optimal scheduler that witnesses the equivalence. We present an example showing that this result also extends to the concurrent setting.

Consider the two contextually equivalent programs in Figure 9. The left-hand side program first non-deterministically chooses a natural number n and samples from rand n. If that result is 0, it loops forever with diverge (), otherwise it directly returns (). The right-hand side program directly returns ().

$$\begin{array}{lcl}\texttt{let } n = \texttt{nondet } () \texttt{ in} & & \\ \texttt{if rand } n = 0 \texttt{ then diverge } () & \simeq_{\mathrm{ctx}} & () \\ \texttt{else } () & & \end{array}$$

Fig. 9. Example without optimal scheduling.

What is the least upper bound of the termination probability of both programs? It is obviously 1 for the right-hand side program, and it turns out it is also 1 for the left-hand side program as well, even though it never terminates with probability 1 for *any* scheduling! This is because for any natural number n, we can find some scheduling such that the left-hand side program terminates with probability $1 - 1/(n+1)$ (by ensuring nondet () returns n). Hence both programs are contextually equivalent!

The left-to-right direction of the refinement is trivial: after assigning a nondeterministic number to n with HT-NONDET, we proceed via a case distinction on whether the sampled value is 0. If yes, we can just apply HT-DIVERGE to obtain a False hypothesis. Otherwise, both programs return (), which satisfy $[\![\mathsf{unit}]\!]_\Delta$.

The right-to-left direction is a bit more involved. Note that we cannot just blindly step through the program because there is no way to continue after arriving at diverge () on the right-hand side. The trick is to avoid arriving at that branch in the first place, but how can that be done? The key idea is that to use PUPD-ERR to produce some positive error $↯(\varepsilon)$ at the beginning. Since $\varepsilon > 0$ there must exists some natural number n such that $\varepsilon > 1/(n+1)$, and hence we have $↯(1/(n+1))$. Now we specifically choose to return n with PUPD-NONDET. When we arrive at the rand sampling instruction, we use PUPD-RAND-EXP for distributing error credits across all the possible branches of

a rand as a weighted sum according to the probability of each branch[8].

$$\frac{\text{PUPD-RAND-EXP}\quad \mathbb{E}_{\mathfrak{U}N}[\mathcal{F}] \le \varepsilon \qquad \lightning(\varepsilon) \qquad j \Mapsto K[\,\mathsf{rand}\,N]}{\pmb{\upsilon}\exists n \le N.\; j \Mapsto K[n] * \lightning(\mathcal{F}(n))}$$

Specifically we choose $\mathcal{F}(n) \triangleq [n = 0]$. That way, when we do a case distinction on whether the sampled value is 0, we can get a contradiction with $\lightning(1)$ in the case we sampled a 0. Otherwise, we return () for both programs, satisfying $[\![\mathsf{unit}]\!]_\Delta$.

D Concurrent Implementation of One-Time Pad

In §3.3, we used presampling tapes to combine randomness arising from concurrent threads. Here, we verify the correctness of a concurrent implementation of a one-time pad, where we use presampling tapes to not combine the randomness of multiple rand operations from concurrent threads, but to linearize them.

A one-time pad is a simple but powerful encryption technique as it is "information-theoretically secure" and cannot be cracked. To encrypt a plaintext, we generate a uniformly-chosen random key and compute the modular addition of the ciphertext and the key. Decryption follows directly by computing the modular addition of the ciphertext and the key.

We consider a simplified program otpImpl in Figure 10, where both the plaintext and key is randomly generated from rand N, but done so in a concurrent way, where the two threads perform a fetch-and-atomic-add instruction into a common reference[9]. We prove its security by showing that the program is contextually equivalent to a rand N (otpSpec):

$$\begin{array}{l}\mathsf{otpImpl} \triangleq \\ \mathsf{let}\ x = \mathsf{ref}\ 0\ \mathsf{in} \\ \left(\begin{array}{l}\mathsf{let\ msg} = \mathsf{rand}\ N\ \mathsf{in} \\ \mathsf{faa}\ x\ \mathsf{msg}\end{array} \;\middle|\middle|\; \begin{array}{l}\mathsf{let\ key} = \mathsf{rand}\ N\ \mathsf{in} \\ \mathsf{faa}\ x\ \mathsf{key}\end{array}\right); \\ !\,x \bmod (N+1)\end{array} \quad \simeq_{\mathrm{ctx}} \quad \begin{array}{l}\mathsf{otpSpec} \triangleq \\ \mathsf{rand}\ N\end{array}$$

Fig. 10. One-time pad example.

Unsurprisingly, the right-to-left direction is not too difficult. Since otpImpl resides on the right-hand side of the refinement, we can take advantage of the angelic nondeterminism of the specification program and choose the interleaving that first resolves the first thread of otpImpl followed by the second. For the sampling of the second thread, we couple it with the one in otpSpec via HT-COUPLE, choosing a bijective function such that the modular addition results in the same value as that returned by otpSpec.

On the other hand, the left-to-right direction is trickier. Morally, depending on how the threads are scheduled, we want to couple the second rand sampling of otpImpl with the rand in otpSpec. However, we do not know in advance which thread is going to be scheduled first when the concurrent program otpImpl is on the left-hand side of our refinement, resulting in a more demonic flavor of nondeterminism.

[8]There is similarly a rule for distributing error credits across a rand for the left-hand side program as well, which we omit for brevity.

[9]Note that the one-time pad is still secure even when the plaintext is not uniformly distributed; we proved a stronger result where the plaintext is generated according to some unknown distribution, and we refer readers to the Rocq repository for more details.

The key idea for proving the refinement from left-to-right is to linearize the random samplings by using tapes to presample the random samplings in advance before the parallel composition in otpImpl. Like before, we prove the refinement via a intermediate program otpImpl′, and it is straightforward to show the refinement between otpImpl and otpImpl′:

$$\begin{array}{l}\mathsf{otpImpl}' \triangleq \mathsf{let}\ x = \mathsf{ref}\ 0\ \mathsf{in}\\ \quad \mathsf{let}\ \kappa = \mathsf{tape}\ N\ \mathsf{in}\\ \quad \mathsf{let}\ \kappa' = \mathsf{tape}\ N\ \mathsf{in}\\ \quad \left(\begin{array}{l}\mathsf{let}\ \mathsf{msg} = \mathsf{rand}\ \kappa\ N\ \mathsf{in}\\ \mathsf{faa}\ x\ \mathsf{msg}\end{array} \;\Big\|\; \begin{array}{l}\mathsf{let}\ \mathsf{key} = \mathsf{rand}\ \kappa'\ N\ \mathsf{in}\\ \mathsf{faa}\ x\ \mathsf{key}\end{array}\right);\\ \quad !\,x \bmod (N+1)\end{array}$$

It then suffices to prove the refinement between otpImpl′ and otpSpec, which turns out to be relatively straightforward as well. After allocating the two tape labels in otpImpl′, we apply PUPD-PRESAMPLE to κ, a simple rule for presampling a number onto the tape on the left-hand side program.

$$\frac{\kappa \hookrightarrow (N, \vec{n})}{\rightsquigarrow (\exists n \le N.\, \kappa \hookrightarrow (N, \vec{n} \cdot n))}\ \textsc{pupd-presample}$$

Then we apply PUPD-COUPLE-LBL-RAND to couple the presampling of the tape κ' and the rand in otpSpec according to the bijective function such that the modular addition of the two presampled values is the same as the one sampled from the specification program. The proof then proceeds by defining an invariant that shares the reference and captures a protocol of how its stored value changes with respect to the threads (using standard Iris ghost resources). After the threads sample the previously presampled value from their respective tapes according to HT-RAND-TAPE, the reference eventually stores the modular addition of both presampled values at the end, matching the one sampled in otpSpec.

It is worth reminding ourselves that the use of presampling tapes in this example is slightly different from that of the batch sampling example in §3.3, where previously, we used tapes to *combine* rand operations, where here, we use them to *linearize* the presampling operations to simplify the coupling argument.

To the best of our knowledge, the verification of otpImpl is out of scope of *all* previous techniques due to its combined use of concurrency, probability, and local state; specifically the probabilistic change in the value of the shared reference is extremely difficult, if not impossible, to capture without ghost resources.

E Algebraic Theory

We consider the algebraic theory induced by contextual equivalence in Foxtrot, summarized in Figure 11.

We first define syntactic sugar for binary nondeterministic (**or**) and probabilistic choice ($\oplus$). To implement nondeterministic choice, we nondeterministically generate a number with nondet (), compare it with 0, and return e_1 or e_2 depending on the result of the test. To implement binary probabilistic choice of some rational bias $0 \le p \le 1$, we sample from rand N and execute e_1 or e_2 depending on whether the value is smaller than M, where N and M are the smallest integers such

that $p = M/(N+1)$.

$$e_1 \,\mathbf{or}\, e_2 \triangleq \mathsf{if}\ \mathsf{nondet}\ () = 0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$$
$$e_1 \oplus_p e_2 \triangleq \mathsf{if}\ \mathsf{rand}\, N < M\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \quad \text{where } M/(N+1) = p$$

The first three equations in Figure 11 show that the probabilistic choice operator satisfies the equational theory of a convex algebra. The next four equations show that the nondeterministic choice operator satisfies the equational theory of a join semilattice.

$$e_1 \oplus_p e_1 \simeq_{\mathrm{ctx}} e_1$$
$$e_1 \oplus_p e_2 \simeq_{\mathrm{ctx}} e_2 \oplus_{1-p} e_1$$
$$(e_1 \oplus_p e_2) \oplus_q e_3 \simeq_{\mathrm{ctx}} e_1 \oplus_{pq} (e_2 \oplus_{\frac{q-pq}{1-pq}} e_3)$$
$$e_1 \,\mathbf{or}\, e_1 \simeq_{\mathrm{ctx}} e_1$$
$$e_1 \,\mathbf{or}\, e_2 \simeq_{\mathrm{ctx}} e_2 \,\mathbf{or}\, e_1$$
$$e_1 \,\mathbf{or}\, (e_2 \,\mathbf{or}\, e_3) \simeq_{\mathrm{ctx}} (e_1 \,\mathbf{or}\, e_2) \,\mathbf{or}\, e_3$$
$$e_1 \,\mathbf{or}\, (\mathsf{diverge}\ ()) \simeq_{\mathrm{ctx}} e_1$$
$$(e_1 \oplus_p e_2) \,\mathbf{or}\, (e_1 \oplus_p e_3) \lesssim_{\mathrm{ctx}} e_1 \oplus_p (e_2 \,\mathbf{or}\, e_3)$$

Fig. 11. Equational theory. Here, $\tau \in \mathit{Type}$ and $\forall i \in \{1,2,3\}.\ \emptyset \vdash e_i : \tau$.

The last equation is the distributive law of a convex semilattice. Currently, Foxtrot is only expressive enough to prove one direction of the equivalence. We hypothesis that the other direction requires more advanced logical facilities for reasoning about probability and nondeterminism such as prophecy variables [1, 2, 22] (as it is the case similarly for an equation in ReLoC [17]) and we leave it as future work.

Almost all of these equations are straightforward to prove; the only exception is that of the third equation, as it involves one to reason about the coupling of a pair of probabilistic choices on both sides, where all four rand operations are sampling from *different* ranges. To reason about this equation, we adopt the key ideas captured by **Slogan 2** (see §3.3), where we introduce several intermediate programs that utilize presampling tapes or parallelize the rand operations, and apply a relatively complex coupling rule that relates all four probabilistic samplings in one go, which we omit for brevity.

F Proof Sketch of Lemma 5.6

We provide a proof sketch of the intermediate result Lemma 5.6 from §5.3 that is used to prove the adequacy theorem Theorem 3.1.

Firstly, we prove Lemma F.1, which informally states that approximate couplings are composable, enabling us to chain couplings of single steps of executions together. This result was first presented in Approxis [19], where the grading ε' can depend on the value of b. We can also derive a similar lemma where one varies the error on A instead, but that is not used in the model of Foxtrot.

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

We also define the trivial FIsch $\Upsilon_\bot$ that only returns None:

Lemma F.2. *There exists a FIsch* $\Upsilon_\bot$ *such that for all* x, *we have* $\mathsf{Flexec}_{\Upsilon_\bot}(x) = \mathrm{ret}\, x$.

We are now ready to provide a proof sketch of Lemma 5.6.

Proof Sketch. The overall proof sketch is similar to a bi-simulation argument. We start by taking induction on the number of steps n taken on the implementation side.

Let us first consider the inductive case. The key idea for the inductive case is that for every action that occurs on the implementation side, i.e. the left-hand side, we extend the FIsch on the specification side, i.e. the right-hand side, such that the extended FIsch on the right continues to satisfy the final approximate coupling goal.

There are four types of actions on the left-hand side to consider during the inductive case:

(1) The left scheduler chooses a thread to step and updates its internal scheduler state
(2) The scheduler erasable state update from the first sstep is applied to the left configuration
(3) The thread chosen takes a step via the pstep
(4) The scheduler erasable state update from the second sstep is applied to the left configuration

As an example, we explain the bi-simulation argument for the second type of action. (We omit the other cases for brevity.) We start by taking induction on the sstep fixpoint, and we outline the proof sketch for the most challenging case spec-step-exp.

Given the premises of spec-step-exp, we first rewrite the left-hand side distribution to be $\mu \gg= (\lambda\, \sigma'.\, \mathrm{exec}_{\zeta,n}(\Xi, (e \cdot \vec{e}, \sigma')))$ with the schErasable condition. It then suffices to find some function f such that $\mathrm{FIschApp}(\Upsilon, f)$ is the oscheduler that satisfies the approximate coupling, where Υ is the FIsch from the inductive hypothesis. We then apply Lemma 5.3 and it suffices to prove the approximate coupling for two "bind" distributions. We apply Lemma F.1 and it suffices to show that there is an approximate coupling between μ and the execution of Υ, which is provided by one of the conditions of spec-step-exp, and for every intermediate result satisfying the relation R, the chosen Υ' according to the function f satisfies the approximate coupling for the $(\lambda\, \sigma'.\, \mathrm{exec}_{\zeta,n}(\Xi, (e \cdot \vec{e}, \sigma')))$ continuation.

Here instead of providing the explicit function f, we apply Lemma 5.5, meaning for each intermediate result that satisfies R, we need to find a single FIsch Υ' that satisfies the approximate coupling. This is done by simply applying the inductive hypothesis of sstep. Here the $\mathsf{stateinj}(R)$ condition is essential because the function f can only depend on the FIsch state on the right hand side, not the configuration on the right hand side or resulting state on the left hand side. The $\mathsf{stateinj}(R)$ condition ensures that the FIsch state Ξ' on the right hand side is unique if there exists states on the left and configurations on the right that satisfies R together with Ξ'.

For the other types of actions, we follow a similar argument as the first sstep in extending the FIsch on the right. The argument is also the same for the base case of the induction. In particular, after "stripping" away one sstep, and we ultimately choose $\Upsilon_\bot$ (Lemma F.2) to be the FIsch on the right that approximately couples the distribution on the left.

□

G Proof of ht-rand

As mentioned in §1.2, with the new definition of the weakest precondition and model, we also have to prove all the proof rules of Foxtrot from scratch, which is *not* automatic. As a proof of concept, we provide a proof sketch of the following rule ht-rand for symbolically executing a rand on the left-hand side program without coupling any execution on the right.

ht-rand

$$\frac{}{\{\mathsf{True}\}\ \mathsf{rand}\, N\ \{n.\, n \in \{0..N\}\}}$$

To do so, we present another way of constructing a FIsch, which intuitively performs one single step, and depending on the outcome, proceeds according to a function f.

Lemma G.1. *There exists a function FIschCons such that given function* $f : Cfg' \to \mathcal{D}(\mathbb{N})$ *and another function* $g : \mathbb{N} \to \mathsf{FIsch}$, *FIschCons*$(f, g)$ *is a FIsch where for all* ρ, *we have* $\mathsf{Flexec}_{FIschCons(f,g)}(\epsilon, \rho) = f(\rho) \gg\!= (\lambda\, j.\, \mathsf{FItpStep}(\rho)(j) \gg\!= (h(j)))$ *where*

$$h(j) \triangleq (\lambda\, \rho'.\, \mathsf{Flexec}_{FIschLift([(\rho,j)],g(j))}([(\rho, j)], \rho'))$$

Lemma G.2. *The rule* HT-RAND *is sound in the model of Foxtrot.*

Proof Sketch. After unfolding the definitions of the weakest precondition, it suffices to use the pstep PROG-STEP-EXP to prove HT-RAND.

The tricky part of this proof is that although we are only stepping on the left-hand side program, we would need to provide a FIsch on the right that couples all the possible changes from the rand expression without taking *any* steps on the program on the right-hand side. The key idea is to stutter the FIsch on the right by choosing to step threads that are outside the range (recall that FItpStep returns the same configuration if we choose to step a thread that does not exist). In particular we choose the FIsch Υ defined as follows for the coupling in pstep:

$$\Upsilon \triangleq \mathsf{FIschCons}((\lambda\, \rho.\, \mathfrak{U}N \gg\!= (\lambda\, j.\ \mathrm{ret}(j + \mathrm{length}(\rho.1)))), (\lambda\, \_.\, \Upsilon_\bot))$$

This FIsch Υ intuitively takes a stutter step by attempting to step the thread $j + \mathrm{length}(\rho.1)$ where j is uniformly chosen from the distribution $\mathfrak{U}N$. Since the thread chosen to step is added as information to the state of the FIsch, we can easily construct a coupling between the updated states of Υ and the result of rand N. Moreover, this updated state of Υ is uniquely defined by the result of rand N so we can easily define a relation R that is state-injective.

□