Almost-Sure Termination by Guarded Refinement

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

Almost-sure termination is an important correctness property for probabilistic programs, and a number of program logics have been developed for establishing it. However, these logics have mostly been developed for first-order programs written in languages with specific syntactic patterns for looping. In this paper, we consider almost-sure termination for higher-order probabilistic programs with general references. This combination of features allows for recursion and looping to be encoded through a variety of patterns. Therefore, rather than developing proof rules for reasoning about particular recursion patterns, we instead propose an approach based on proving *refinement* between a higher-order program and a simpler probabilistic model, in such a way that the refinement preserves termination behavior. By proving a refinement, almost-sure termination behavior of the program can then be established by analyzing the simpler model.

We present this approach in the form of Caliper, a higher-order separation logic for proving termination-preserving refinements. Caliper uses probabilistic couplings to carry out relational reasoning between a program and a model. To handle the range of recursion patterns found in higher-order programs, Caliper uses guarded recursion, in particular the principle of Löb induction. A technical novelty is that Caliper does not require the use of transfinite step indexing or other technical restrictions found in prior work on guarded recursion for termination-preservation refinement. We demonstrate the flexibility of this approach by proving almost-sure termination of several examples, including first-order loop constructs, a random list generator, treaps, and a sampler for Galton-Watson trees that uses higher-order store. All the results have been mechanized in the Coq proof assistant.

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

Additional Key Words and Phrases: almost-sure termination, probabilistic coupling, Markov chains

ACM Reference Format:
Simon Oddershede Gregersen, Alejandro Aguirre, Philipp G. Haselwarter, Joseph Tassarotti, and Lars Birkedal. 2024. Almost-Sure Termination by Guarded Refinement. *Proc. ACM Program. Lang.* 8, ICFP, Article 243 (August 2024), 31 pages. https://doi.org/10.1145/3674632

[*]The majority of this work was carried out while the author was affiliated with Aarhus University.

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

1 Introduction

Probabilistic programs are programs that draw samples from probability distributions. They have many applications but also complex and unintuitive behaviors. Therefore, there has been long-standing interest in formal techniques for reasoning about them.

In particular, termination of probabilistic programs is an important property for the correctness of various sampling and consensus algorithms. This work considers the problem of *almost-sure termination* (AST), that is, whether a probabilistic program terminates with probability 1. The problem of showing AST is known to be computationally harder than termination of deterministic programs [Kaminski et al. 2019] so there is clearly no hope of completeness. On the other hand, the problem is decidable for large classes of probabilistic processes [Brázdil et al. 2013], and indeed such decision procedures have been implemented in probabilistic model checkers such as PRISM [Kwiatkowska et al. 2011]. Moreover, there is a long and rich history in the mathematical literature that has studied the termination behavior of multiple families of stochastic processes [Athreya and Ney 1972; Spitzer 1964].

Multiple works have developed program logics to reason about termination or expected running time of probabilistic programs. For example, the work on weakest pre-expectation calculi by Morgan and McIver [1999] can, in particular, be used to prove AST, and the expected runtime transformer by Kaminski et al. [2016] provides a compositional way to reason about the running time of probabilistic programs. Ranking Supermartingales [Chakarov and Sankaranarayanan 2013; Fu and Chatterjee 2019] are probabilistic analogues of ranking functions that can be used to prove termination of probabilistic programs by adapting results from martingale theory to program verification. Others [McIver et al. 2018] develop rules to prove termination of particularly complex iteration schemes.

The works listed above have all increased the reach of termination analysis of probabilistic programs, but they are mostly limited to first-order, imperative languages and their techniques are adapted to this particular setting. Some works [Avanzini et al. 2021; Kobayashi et al. 2019] have considered stateless higher-order programs and higher-order recursion schemes, but it is unclear to what extent these approaches scale to richer languages. For instance, in a setting that includes higher-order functions and higher-order references, recursion and thus divergence can arise in multiple ways, *e.g.*, by recursion through the store, so rules that have been tailored to syntactic while loops or particular recursion schemes will be difficult to generalize and apply.

As an example, consider the randomized function walk shown below, which uses a fixed-point combinator fix defined using Landin's knot [Landin 1964]:

$$\begin{aligned}
\mathsf{fix} &\triangleq \lambda f.\,\mathsf{let}\ r = \mathsf{ref}\ (\lambda x.\,x)\ \mathsf{in}\ r \leftarrow (\lambda x.\,f\ (!\,r)\ x);\ !\,r \\
\mathsf{F} &\triangleq \lambda f.\,\lambda n.\,\mathsf{if}\ n == 0\ \mathsf{then}\ () \\
&\qquad\qquad \mathsf{else\ if}\ \ \mathsf{flip}\ \ \mathsf{then}\ f\ (n-1)\ \mathsf{else}\ f\ (n+1) \\
\mathsf{walk} &\triangleq \mathsf{fix}\ \mathsf{F}
\end{aligned}$$

The flip expression in F reduces uniformly at random to either true or false.

The execution of walk n depends on a complex interaction between probabilistic choice and higher-order store. Nevertheless, walk n almost-surely terminates for any starting value n. The reason is that the value of the argument on each successive call to F follows a symmetric random walk on the natural numbers which terminates upon reaching 0. This is a well-known stochastic process that almost-surely terminates. The random walk can be represented by the probabilistic transition system depicted below

$$\begin{tikzcd}
0 & 1 \arrow[l, bend left, "\frac{1}{2}"] \arrow[r, bend left, "\frac{1}{2}"] & 2 \arrow[l, bend left, "\frac{1}{2}"] \arrow[r, bend left, "\frac{1}{2}"] & 3 \arrow[l, bend left, "\frac{1}{2}"] \arrow[r, bend left, "\frac{1}{2}"] & \cdots \arrow[l, bend left, "\frac{1}{2}"]
\end{tikzcd}$$

where, in this analogy, the labels on the states represent the current value of the argument n, and 0 represents a terminal state, corresponding to the fact that recursion stops when n is 0. Since execution of walk n terminates whenever the transition system terminates, and the transition system almost-surely terminates, walk n must almost-surely terminate.

In this work, we explore a novel approach to proving AST that allows us to capture this kind of argument in a precise and formal way. We develop Caliper, a *higher-order guarded separation logic* that allows one to prove termination of examples like walk n. Instead of reasoning about the termination probability directly, Caliper establishes a termination-preserving *refinement* between a user-chosen probabilistic model (like the random walk above) and a probabilistic program (such as walk n). Termination preservation here implies that the probability of termination of the program is at least as high as the probability of termination of the model. Thus, if the model almost-surely terminates, so does the program. This allows us to transfer the problem of proving AST of a probabilistic program to that of proving AST of a model, and in turn it allows us to make use of a wide set of tools for showing termination of the model. The benefit of this approach is that it makes it possible to apply the rich theory and extensive prior work that has been developed for AST of first-order programs, without the need to adapt that work to the setting of a higher-order language. In particular, for the motivating example at hand, we have used Caliper to show that walk n refines the symmetric random walk model and, as a consequence, that it almost-surely terminates.

To support reasoning about the different forms of recursion present in higher-order languages with higher-order store, Caliper uses *guarded recursion* based on step indexing [Appel et al. 2007; Birkedal et al. 2012; Nakano 2000]. Several recent studies have used guarded recursion for termination-preserving refinement for *non*-probabilistic programs [Spies et al. 2021a; Tassarotti et al. 2017; Timany et al. 2024a], but it is not *a priori* clear that their formulation of refinement can be adapted to the probabilistic setting. In fact, to preserve termination, that prior work has had to impose various restrictions on non-determinism of programs, or to replace standard step indexing with *transfinite* step indexing, and so one might expect that probabilistic termination preservation would require similar technical changes. Surprisingly, as we show in §6, it turns out that these workarounds are not needed in the probabilistic setting.

One limitation of the refinement approach is that it requires coming up with suitable models of programs. If the model is very close to the original program, the refinement may be easy to show, yet analyzing the termination of the model is then no simpler. On the other hand, if the model is considerably simpler than the program, one might worry that the intended refinement is difficult to prove. To demonstrate empirically that Caliper is effective, we have verified a range of examples, several of which demonstrate intricate use of higher-order functions and higher-order state, putting them beyond the scope of previous techniques.

Contributions. In summary,

(1) We develop a compositional, higher-order guarded separation logic for showing termination-preserving refinement of higher-order probabilistic programs, which we use as a technique for showing almost-sure termination.
(2) We identify two new and orthogonal uses cases for *asynchronous coupling* [Gregersen et al. 2024b] as a mechanism for (a) coupling *one* model step to *multiple* program samplings, and (b)

managing guarded recursion, that is, to eliminate later modalities *now*, which could otherwise only have been eliminated *in the future* if ordinary couplings had been used.

(3) We demonstrate our approach on a rich set of examples, showcasing the potential both on the "classic" first-order examples but also on more involved implementations which make use of local and dynamically-allocated higher-order state. Our examples also demonstrate how our approach allows for concise, composable, and higher-order specifications that resemble specifications in non-probabilistic separation logics.

(4) All of the results presented in this paper are mechanized [Gregersen et al. 2024a] in the Coq proof assistant [Coq Development Team 2023], including the semantics, the logic, the mathematical analysis results, and the case studies, with the help of the Coquelicot library for real analysis [Boldo et al. 2015] and the Iris separation logic framework [Jung et al. 2018].

2 Background and Preliminaries

In this section, we recall some basic definitions from probability theory and we define what it means to execute a probabilistic program. Although the Caliper approach is language generic, in this paper we fix a probabilistic ML-like language, whose semantics we describe here. Finally, we define a notion of probabilistic coupling that will be central to the soundness of our approach.

2.1 Probabilistic Semantics

To account for non-terminating behavior, we make use of (discrete) probability *sub*-distributions.

Definition 2.1. A *sub-distribution* over a countable set A is a function $\mu : A \to [0,1]$ such that $\sum_{a \in A} \mu(a) \leq 1$. We write $\mathcal{D}(A)$ for the set of all sub-distributions over A.

Definition 2.2. The *support* of $\mu \in \mathcal{D}(A)$ is the set of elements $\text{supp}(\mu) \triangleq \{a \in A \mid \mu(a) > 0\}$.

Lemma 2.3. *Let $\mu \in \mathcal{D}(A)$, $a \in A$, and $f : A \to \mathcal{D}(B)$. Then*

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

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

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

We obtain a small-step operational semantics for both programs and models by considering them as (discrete-time) *Markov chains*.

Definition 2.4. A (sub)-*Markov chain* over a countable set M is a function $\text{step} \colon M \to \mathcal{D}(M)$.

Given a Markov chain over M and a decidable predicate $\text{final} : M \to \textit{Prop}$ such that if $\text{final}(m)$ then $\text{step}(m)(m') = 0$ for all m', we define what it means for a Markov chain to evaluate to a final state. First, we define a stratified execution distribution $\text{exec}_n \colon M \to \mathcal{D}(M)$ by induction on n:

$$\text{exec}_n(m) \triangleq \begin{cases} \mathbf{0} & \text{if } \neg\, \text{final}(m) \text{ and } n = 0 \\ \text{ret}(m) & \text{if } \text{final}(m) \\ \text{step}(m) \gg\!= \text{exec}_{(n-1)} & \text{otherwise} \end{cases}$$

where $\mathbf{0}$ denotes the everywhere-zero sub-distribution. Observe that the value $\text{exec}_n(m)(m')$ denotes the probability of stepping from a state m to a final state m' in at most n steps. The probability that an execution starting from a state m reaches a final state m' is the limit of its stratified approximations, which exists by monotonicity and boundedness:

$$\text{exec}(m)(m') \triangleq \lim_{n \to \infty} \text{exec}_n(m)(m').$$

The probability that an execution from state m terminates is thus $\mathrm{exec}_{\Downarrow}(m) \triangleq \sum_{m' \in M} \mathrm{exec}(m)(m')$. By the monotone convergence theorem we get the lemma below, which intuitively says that it suffices to consider all finite approximations to bound the termination probability.

Lemma 2.5. *If* $\sum_{m' \in M} \mathrm{exec}_n(m)(m') \leq r$ *for all* n *then* $\mathrm{exec}_{\Downarrow}(m) \leq r$.

ProbLang. The syntax of ProbLang, the programming language we consider throughout this paper, is defined by the grammar below.

$$\begin{aligned}
v, w \in \mathit{Val} &::= z \in \mathbb{Z} \mid b \in \mathbb{B} \mid () \mid \ell \in \mathit{Loc} \mid \mathsf{rec}\, f\ x = e \mid (v, w) \mid \mathsf{inl}\, v \mid \mathsf{inr}\, v \mid \\
e \in \mathit{Expr} &::= v \mid x \mid e_1 + e_2 \mid e_1 - e_2 \mid \ \ldots\ \mid e_1\ e_2 \mid \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e \mid \\
&\qquad \mathsf{match}\ e\ \mathsf{with}\ \mathsf{inl}\, v \Rightarrow e_1 \mid \mathsf{inr}\, w \Rightarrow e_2\ \mathsf{end} \mid \mathsf{ref}\, e \mid\ !\, e \mid e_1 \leftarrow e_2 \mid \mathsf{rand}\, e \\
K \in \mathit{Ectx} &::= - \mid e\, K \mid K\, v \mid \mathsf{ref}\, K \mid\ !\, K \mid e \leftarrow K \mid K \leftarrow v \mid \mathsf{rand}\, K \mid \ldots \\
\sigma \in \mathit{State} &\triangleq \mathit{Loc} \xrightarrow{\text{fin}} \mathit{Val} \\
\rho \in \mathit{Cfg} &\triangleq \mathit{Expr} \times \mathit{State}
\end{aligned}$$

The term language is mostly standard: $\mathsf{ref}\, e$ allocates a new reference, $!\, e$ dereferences the location e evaluates to, and $e_1 \leftarrow e_2$ assigns the result of evaluating e_2 to the location that e_1 evaluates to. We introduce syntactic sugar for lambda abstractions $\lambda x.\, e$ defined as $\mathsf{rec}\ \_\ x = e$, let-bindings $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ defined as $(\lambda x.\, e_2)\ e_1$, and sequencing $e_1; e_2$ defined as $\mathsf{let}\ \_ = e_1\ \mathsf{in}\ e_2$.

The language has a call-by-value Markov chain semantics $\mathrm{step} : \mathit{Cfg} \to \mathcal{D}(\mathit{Cfg})$ defined using evaluation contexts $K \in \mathit{Ectx}$. We set $\mathrm{final}(e, \sigma) \triangleq (e \in \mathit{Val})$. The semantics is mostly standard: all the non-probabilistic constructs reduce as usual with weight 1, *e.g.*, $\mathrm{step}(\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2, \sigma) = \mathrm{ret}(e_1, \sigma)$ and $\mathsf{rand}\, N$ reduces uniformly at random, *i.e.*,

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

We recover the Boolean operation flip by defining $\mathsf{flip} \triangleq (\mathsf{rand}\, 1 == 1)$.

2.2 Probabilistic Couplings

Probabilistic coupling [Lindvall 2002; Thorisson 2000; Villani 2009] is a mathematical technique for reasoning about pairs of probabilistic processes. Informally, couplings relate the outputs of two processes by specifying how corresponding sampling statements should be correlated. This correlation is described by constructing a particular joint distribution over pairs of samples from the two processes. Traditional definitions of couplings implicitly require that the masses of the two distributions being related are the same. Instead, we make use of an *asymmetric* notion of couplings, which allows the left-hand side distribution to have less mass than the right-hand side distribution.

Definition 2.6 (Left-partial coupling [Gregersen et al. 2024b])**.** Let $\mu_1 \in \mathcal{D}(A)$ and $\mu_2 \in \mathcal{D}(B)$. A sub-distribution $\mu \in \mathcal{D}(A \times B)$ is a *left-partial coupling* of μ_1 and μ_2 if

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

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

Notice how $\mathbf{0} \lesssim \mu : R$ holds trivially for any $\mu \in \mathcal{D}(B)$ and $R \subseteq A \times B$ by picking $\mathbf{0} \in \mathcal{D}(A \times B)$ as the witness. This would not be the case for the traditional symmetric definition of coupling.

Once a coupling has been established, we can often use it to extract a concrete relation between the two probability distributions. *E.g.*, for $(=)$-couplings, we can conclude point-wise inequality.

Lemma 2.7. *If $\mu_1 \lesssim \mu_2 : (=)$ then $\mu_1(a) \leq \mu_2(a)$ for all a.*

Moreover—most important for our purposes—from *any* left-partial coupling of μ_1 and μ_2, we can conclude that the mass of μ_1 bounds the mass of μ_2 from below.

Lemma 2.8. *If $\mu_1 \lesssim \mu_2$ then $\sum_{a \in A} \mu_1(a) \leq \sum_{b \in B} \mu_2(b)$.*

As part of the proof of our soundness theorem (Theorem 3.1), we show that the refinement logic constructs a left-partial coupling of a partial execution of the model and the full execution of the program. Using Lemma 2.5 and Lemma 2.8 we may then conclude that the termination probability of the program is bounded below by the termination probability of the model.

3 A Probabilistic Termination-Preserving Refinement Logic

Caliper is a *probabilistic relational separation logic*. In this section, we give a high-level overview of Caliper's rules and walk through simple example uses.

Caliper uses a *refinement* weakest precondition, written $\mathsf{rwp}\ e\ \{\Phi\}$, where e is the program we want to prove a refinement about and Φ is a postcondition. As in much prior work on refinement reasoning in separation logic, *e.g.*, Gregersen et al. [2024b]; Timany et al. [2024b]; Turon et al. [2013], the model that we want to relate to e is tracked as a *ghost state* assertion of the form $\mathsf{spec}(m)$, which asserts that the model is currently in state m. We use Markov chains to represent model systems.

To establish a termination-preserving refinement between a Markov chain model starting in a state m and a program e, we prove an entailment of the form $\mathsf{spec}(m) \vdash \mathsf{rwp}\ e\ \{\Phi\}$, for an arbitrary postcondition Φ. The following soundness theorem then implies a lower bound between termination of the model and the program:

Theorem 3.1 (Soundness). *Let m be a state of a Markov chain. If*

$$\mathsf{spec}(m) \vdash \mathsf{rwp}\ e\ \{\Phi\}$$

then $\mathsf{exec}_{\Downarrow}(m) \leq \mathsf{exec}_{\Downarrow}(e, \sigma)$ for all program states σ.

For the motivating example discussed in §1, if we label the states of the random walk Markov chain by numbers n, then showing $\mathsf{spec}(n) \vdash \mathsf{rwp}\ \mathsf{walk}\ n\ \{\mathsf{True}\}$ will establish that $\mathsf{walk}\ n$ is a termination-preserving refinement of the model from starting state n. Because the model almost-surely terminates for all n, we thus get that $\mathsf{walk}\ n$ almost-surely terminates.

3.1 Separation Logic Connectives and Basic Unary Rules

Caliper is developed on top of the Iris framework [Jung et al. 2018] and inherits Iris's basic separation logic connectives. We write $P * Q$ for *separating conjunction*, $P \mathrel{-\!\!*} Q$ for its adjoint *separating implication* (magic wand), and $\ell \mapsto v$ for the separation logic resource that denotes ownership of the location ℓ containing the value v. Throughout the paper we omit connectives that are used to manipulate Iris-style ghost resources and invariants, *e.g.*, the *fancy update modality* [Jung et al. 2018] of Iris, since these ideas are orthogonal to the core challenge of refinement reasoning and our use is entirely standard.

For proving refinement weakest preconditions, the logic has typical separation logic rules for reasoning about the basic commands of the language. A selection of these rules is shown in Figure 1. The relation $e_1 \rightsquigarrow e_2$ says that e_1 can take a *pure* step, *i.e.*, a non-stateful and non-probabilistic reduction step, to the expression e_2.

3.2 Guarded Recursion and Relational Rules

Notably, there is no rule in Figure 1 for reasoning about recursion or loops. As alluded to in §1, there are multiple ways to encode a form of recursion in a language like ProbLang, so rules based

$$e_1 \rightsquigarrow e_2 * \mathsf{rwp}\ e_2\ \{\Phi\} \vdash \mathsf{rwp}\ e_1\ \{\Phi\} \qquad \textsc{rwp-pure}$$

$$\forall \ell.\, \ell \mapsto v \mathrel{-\!\!*} \Phi(\ell) \vdash \mathsf{rwp}\ \mathsf{ref}\, v\ \{\Phi\} \qquad \textsc{rwp-alloc}$$

$$(\ell \mapsto v \mathrel{-\!\!*} \Phi(v)) * \ell \mapsto v \vdash \mathsf{rwp}\ !\,\ell\ \{\Phi\} \qquad \textsc{rwp-load}$$

$$(\ell \mapsto w \mathrel{-\!\!*} \Phi()) * \ell \mapsto v \vdash \mathsf{rwp}\ \ell \leftarrow w\ \{\Phi\} \qquad \textsc{rwp-store}$$

$$\forall n \leq N.\, \Phi(n) \vdash \mathsf{rwp}\ \mathsf{rand}\, N\ \{\Phi\} \qquad \textsc{rwp-rand}$$

$$\Phi(v) \vdash \mathsf{rwp}\ v\ \{\Phi\} \qquad \textsc{rwp-val}$$

$$\mathsf{rwp}\ e\ \big\{v.\, \mathsf{rwp}\ K[v]\ \{\Phi\}\big\} \vdash \mathsf{rwp}\ K[e]\ \{\Phi\} \qquad \textsc{rwp-bind}$$

$$(\forall v.\, \Psi(v) \mathrel{-\!\!*} \Phi(v)) * \mathsf{rwp}\ e\ \{\Psi\} \vdash \mathsf{rwp}\ e\ \{\Phi\} \qquad \textsc{rwp-mono}$$

$$P * \mathsf{rwp}\ e\ \{\Phi\} \vdash \mathsf{rwp}\ e\ \{v.\, P * \Phi(v)\} \qquad \textsc{rwp-frame}$$

Fig. 1. Program logic rules governing the rwp e $\{\Phi\}$ connective.

$$\textsc{later-intro}\ \frac{P \vdash Q}{P \vdash \triangleright Q} \qquad \textsc{later-mono}\ \frac{P \vdash Q}{\triangleright P \vdash \triangleright Q} \qquad \textsc{later-and}\ \frac{P \vdash \triangleright(Q \wedge R)}{P \vdash \triangleright Q \wedge \triangleright R} \qquad \textsc{later-sep}\ \frac{P \vdash \triangleright(Q * R)}{P \vdash \triangleright Q * \triangleright R}$$

Fig. 2. Selected rules for the $\triangleright$ modality.

on specific syntactic patterns cannot cover the full range of such mechanisms. Instead, Caliper makes use of *guarded* recursion.

To explain and motivate the need for guarded recursion, let us return to the random walk example from §1 and see the issues that arise in trying to prove spec$(n) \vdash$ rwp walk n {True}.

One might first try to prove this by induction on n. However, this attempt would fail in the inductive case, since when the flip in walk n resolves to false, the code effectively makes a recursive call in which the argument is *incremented* to $n + 1$. Thus, in that branch, after stepping through the definition of *fix* and F, we will eventually find ourselves having to prove a goal of the form spec$(n + 1) \vdash$ rwp walk $(n + 1)$ {True}, which does not match the induction hypothesis.

Instead, the solution in Caliper is to make use of guarded recursion, in particular the Löb induction rule [Nakano 2000]:

$$\textsc{löb}\ \frac{\triangleright P \vdash P}{\mathsf{True} \vdash P}$$

which says that to prove P, it suffices to prove P under the induction hypothesis $\triangleright P$, where $\triangleright$ is the so-called "later" modality [Appel et al. 2007; Birkedal et al. 2012; Nakano 2000]. Selected other rules for this modality are shown in Figure 2. For our example, taking P in the Löb rule to be $(\forall n.\ \mathsf{spec}(n) \mathrel{-\!\!*} \mathsf{rwp\ walk}\, n\ \{\mathsf{True}\})$, will mean that this induction hypothesis allows us to assume the desired refinement holds for *all* n, albeit guarded by the later modality. Because the hypothesis applies for all n, we will not run into the obstacle we had with induction on n when the branch in flip resolved to false.

But how do we eliminate the $\triangleright$ modality guarding the induction hypothesis? In most guarded program logics, the later modality can be eliminated whenever we take a "step" of the program being verified. However, following Spies et al. [2021a], in Caliper the later modality may only be eliminated when the *model* program performs a transition (more precisely, when the model transition system has a transition from the current state of the model to another state). Since Caliper has no other built-in rule for reasoning about loops or recursion, this ensures that each time the program does some kind of looping or recursion using the LÖB rule, at least one step will have been performed by the model. Intuitively, this means that the program can only have a non-terminating execution if the model can take infinitely many transitions.

The simplest later elimination rule applies when the model can make a deterministic transition:

$$\textsc{rwp-spec-det}\quad \frac{\mathsf{step}(m_1)(m_2) = 1 \qquad \mathsf{spec}(m_2) * P \vdash \mathsf{rwp}\ e\ \{\Phi\}}{\mathsf{spec}(m_1) * \triangleright P \vdash \mathsf{rwp}\ e\ \{\Phi\}}$$

If the model is currently in a state m_1, as witnessed by ownership of the resource $\mathsf{spec}(m_1)$, and the model can deterministically make a step to m_2, then we may progress the model, stripping a later modality from the assumption P.

3.3 Coupling Rules

Of course, RWP-SPEC-DET is not sufficient for the example at hand, in which the model only has randomized transitions. To address this, Caliper also satisfies the following *coupling* rule, similar to that of pRHL [Barthe et al. 2008], in which the possible transitions of the model and program must be "matched up" in a way that preserves probabilities:

$$\textsc{rwp-coupl-rand}\quad \frac{\mathsf{step}(m_1) \lesssim \mathsf{unif}(N) : R \qquad \vdash \forall (m_2, n) \in R.\, (\mathsf{spec}(m_2) * P) \mathrel{-\!\!*} \mathsf{rwp}\ n\ \{\Phi\}}{\mathsf{spec}(m_1) * \triangleright P \vdash \mathsf{rwp}\ \ \mathsf{rand}\, N\ \{\Phi\}}$$

where $\mathsf{unif}(N)$ denotes the uniform distribution on the set $\{0, 1, \ldots N\}$. When executing a rand N command, if the model is currently in the state m_1, the rule says that if we can show a probabilistic coupling $\mathsf{step}(m_1) \lesssim \mathsf{unif}(N) : R$ of the two steps, then we may continue reasoning *as if* the program and the model progressed to states in the support of the coupling. Furthermore, the $\triangleright$ guarding the assumption P is removed, reflecting that the model has made a transition.

As an example, a special case of this rule for flip is the following:

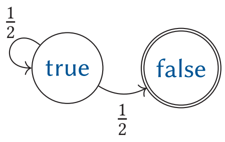
$$\frac{\begin{array}{cc} & m_f \neq m_t \\ \mathsf{step}(m)(m_f) = 1/2 & P * \mathsf{spec}(m_f) \vdash \mathsf{rwp}\ K[\mathsf{false}]\ \{\Phi\} \\ \mathsf{step}(m)(m_t) = 1/2 & P * \mathsf{spec}(m_t) \vdash \mathsf{rwp}\ K[\mathsf{true}]\ \{\Phi\} \end{array}}{\triangleright P * \mathsf{spec}(m) \vdash \mathsf{rwp}\ K[\mathsf{flip}]\ \{\Phi\}}$$

With this rule, we have to reason about two cases for how the flip command resolves, and for each case, we pick a state to transition the model to, subject to the requirement that the model transitions happen with the same probability 1/2 as the transitions that flip makes.

In the walk example, when the model is in state $m = n$, we take $m_f = n + 1$ and $m_t = n - 1$. Thus, for the case where the flip resolves to false and then makes the recursive call of walk $(n + 1)$, we will have $\mathsf{spec}(n + 1)$. Therefore, we may make use of the induction hypothesis from Löb induction, which will have had the $\triangleright$ modality removed through the application of the rule for flip. The case for true is similar.

3.4 Example: Repeated Coin Flips

We now consider another example of reasoning about loops in Caliper. One of the simplest almost-surely (but not always) terminating stochastic processes is the one that repeatedly tosses a fair coin until it gets tails. There exists a non-terminating run, *i.e.*, the run where one always gets heads, but it happens only with probability zero. We can describe the process using the following diagram where true is used for heads and false for tails.



It is straightforward to show that $\text{exec}_n(\text{true}) = 1 - \frac{1}{2}^n$ and thus $\text{exec}_{\Downarrow}(\text{true}) = 1$.

Using the notation $\text{while } e_1 \text{ do } e_2 \text{ end} \triangleq (\text{rec } f\ () = \text{if } e_1 \text{ then } e_2; f\ () \text{ else } ())\ ()$ we implement this process as a loop that repeatedly flips a coin until it gets false.

$$\text{flips} \triangleq \text{while flip do } () \text{ end}$$

Using Caliper, we show that flips formally refines the model by showing the specification

$$\vdash \text{spec}(\text{true}) \twoheadrightarrow* \text{rwp flips } \{\_.\ \text{spec}(\text{false})\}$$

and thus flips almost-surely terminates.

To prove the entailment we apply LÖB and are left with the proof obligation

$$\triangleright\left(\text{spec}(\text{true}) \twoheadrightarrow* \text{rwp flips } \{\_.\ \text{spec}(\text{false})\}\right) \vdash \text{spec}(\text{true}) \twoheadrightarrow* \text{rwp flips } \{\_.\ \text{spec}(\text{false})\}.$$

That is, we have assumed our initial goal but under a later modality. After introducing the specification resource, we symbolically step the program forward using RWP-PURE. We now apply RWP-COUPL-RAND using an equality coupling of unif(1) which allows us to strip the later modality from the induction hypothesis and continue reasoning *as if* the coin in the program and the model have the same outcome b.

$$\begin{aligned}&\text{spec}(b) * \left(\text{spec}(\text{true}) \twoheadrightarrow* \text{rwp flips } \{\_.\ \text{spec}(\text{false})\}\right) \vdash \\ &\text{rwp if } b \text{ then } ();\ \text{flips else } () \ \{\_.\ \text{spec}(\text{false})\}.\end{aligned}$$

We continue by case distinction on b: if b is false we are immediately done and if b is true we apply RWP-PURE followed by the induction hypothesis which finishes the proof.

While the repeated coin flip is a simple example that mainly serves to illustrate a minimal proof in Caliper, the same pattern and recipe applies to many other first-order probabilistic looping constructs as well. For example, the program

```
let r = ref n in
while !r ≠ 0 do
  if flip then r ← !r − 1 else r ← !r + 1
end
```

can be shown to refine the symmetric random walk from §1 using exactly this pattern and the refinement proof looks much like for the repeated coin flip.

3.5 Summary

As we have seen, at a high level, the rules of Caliper combine three key ingredients:

(1) *Higher-order separation logic*, as embodied in the Iris framework [Jung et al. 2018], which provides powerful tools for reasoning about modular use of state in higher-order programs.
(2) *Guarded recursion*, formulated as in recent works on termination-preserving refinement [Spies et al. 2021a] to ensure that looping in the program is matched with transitions in the model.
(3) *Couplings*, as in pRHL [Barthe et al. 2008], which allow for "aligning" the probabilistic transitions of the program and model.

In the end, the logic may seem surprisingly—and perhaps suspiciously—simple, but this simplicity stems from the use of these powerful and well-tested abstractions. The main challenge and novelty of Caliper, then, lies in showing that this combination of rules is sound for proving probabilistic termination-preserving refinements, the question that we will turn to in §6. Before doing so, we first examine a way to make the coupling rule more flexible (§4) and then explore a number of case studies using Caliper (§5).

4 Asynchronous Couplings

Using the probabilistic coupling rules we have seen so far requires aligning or "synchronizing" the sampling statements of the two probabilistic processes being related. For example, both the program and its model have to be executing the sample statements we want to couple for their next step when applying rules like RWP-COUPL-RAND. However, it is not always possible to synchronize sampling statements in this way, especially when considering higher-order programs. To address this issue, Gregersen et al. [2024b] introduce *asynchronous coupling* for proving contextual refinement of (higher-order) probabilistic programs. We identify two new and orthogonal use cases for asynchronous coupling in Caliper which address the following two issues:

(1) When relating a complex program to a simpler model, it is sometimes necessary to couple *one* model step to *multiple* non-adjacent program samplings (as illustrated in §5.3).
(2) Sometimes a later modality needs to be eliminated *now*, but the coupling step—which would introduce the later modality—only happens *in the future* (as illustrated in §5.5).

In the remainder of this section, we recall the concept of asynchronous coupling and describe how it is incorporated into Caliper. The reader may want to initially skip this section but return before reading §5.3 and §5.5.

Presampling tapes. Asynchronous couplings are introduced through dynamically-allocated *presampling tapes*. Intuitively, presampling tapes will allow us *in the logic* to presample (and in turn couple) the outcome of future sampling statements.

Formally, presampling tapes appear as two new constructs added to the programming language.

$$e \in \mathit{Expr} ::= \ \dots \ \mid \ \mathsf{tape}\ \ e \mid \ \mathsf{rand}\, e_1\ e_2$$

The $\mathsf{tape}\ \ N$ operation allocates a new fresh tape with the upper bound N, representing future outcomes of $\mathsf{rand}\, N$ operations. The rand primitive can now (optionally) be annotated with a tape label ι. If the corresponding tape is empty, $\mathsf{rand}\ \ N\ \iota$ reduces to any $n \leq N$ with equal probability, just as if it had not been labeled. But if the tape is *not* empty, then $\mathsf{rand}\ \ N\ \iota$ reduces *deterministically* by taking off the first element of the tape and returning it. However, *no* primitives in the language will add values to the tapes. Instead, values are added to tapes as part of presampling steps that will be *ghost operations* appearing only in the logic. In fact, labeled and unlabeled samplings operations are contextually equivalent [Gregersen et al. 2024b].

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

$$\forall \iota.\, \iota \hookrightarrow (N, \epsilon) \mathrel{-\!\!*} \Phi(\iota) \vdash \mathsf{rwp}\ \ \mathsf{tape}\, N\, \{\Phi\} \qquad \text{RWP-TAPE-ALLOC}$$

If one owns $\iota \hookrightarrow (N, \epsilon)$, *i.e.*, when the corresponding tape is empty, then $\mathsf{rand}\ \ N\ \iota$ reduces symbolically to any $n \leq N$, reflecting the operational behavior described above:

$$(\forall n \leq N.\, \iota \hookrightarrow (N, \epsilon) \mathrel{-\!\!*} \Phi(n)) * \iota \hookrightarrow (N, \epsilon) \vdash \mathsf{rwp}\ \ \mathsf{rand}\, N\, \iota\, \{\Phi\} \qquad \text{RWP-TAPE-EMPTY}$$

When the tape is *not* empty, then rand $N\ \iota$ reduces symbolically by taking off the first element of the tape and returning it.

$$(\iota \hookrightarrow (N, \vec{n}) \mathrel{-\!\!*} \Phi(n)) * \iota \hookrightarrow (N, n \cdot \vec{n}) \vdash \mathsf{rwp}\ \mathsf{rand}\, N\ \iota\, \{\Phi\} \qquad \text{RWP-TAPE}$$

Asynchronous couplings can now be introduced in the logic by coupling rules that couple *any* finite number of presampling steps onto tapes with a model step. When we—at some point in the future—reach a presampled rand $N\ \iota$ operation, we simply read off the presampled values from the ι tape deterministically in a first-in-first-out order. For example, an asynchronous variant of the coupling rule for flip originally shown in §3 is the following:

$$\frac{\begin{array}{cc} & m_f \neq m_t \\ \mathsf{step}(m)(m_f) = 1/2 & P * \mathsf{spec}(m_f) * \iota \hookrightarrow (1, \vec{b} \cdot \mathsf{false}) \vdash \mathsf{rwp}\ e\ \{\Phi\} \\ \mathsf{step}(m)(m_t) = 1/2 & P * \mathsf{spec}(m_t) * \iota \hookrightarrow (1, \vec{b} \cdot \mathsf{true}) \vdash \mathsf{rwp}\ e\ \{\Phi\} \end{array}}{\triangleright P * \iota \hookrightarrow (1, \vec{b}) * \mathsf{spec}(m) \vdash \mathsf{rwp}\ e\ \{\Phi\}}$$

Instead of reasoning about two cases for how a flip operation resolves, we reason about two cases for how a Boolean is sampled onto the tape ι. This pattern can be generalized to allow one model step to be coupled with multiple presampling steps, which we exploit in §5.3. Notice moreover that since a model step is taken, we are also allowed to eliminate a later modality from the assumption P *before* reaching the flip operation. This fact will be crucial for the example considered in §5.5.

Soundness of asynchronous couplings hinges on the fact that presampling operationally *does not matter* as seen by the erasure theorem below. The distribution $\mathsf{sstep}(\sigma_1, \iota)$ appends a uniformly sampled value to the end of the ι tape in σ_1.

Lemma 4.1 (Erasure). *If* $\iota \in \mathsf{dom}(\sigma_1)$ *then* $\mathsf{exec}(e, \sigma_1) = \mathsf{sstep}(\sigma_1, \iota) \gg\!= (\lambda \sigma_2.\ \mathsf{exec}(e, \sigma_2))$.

5 Case Studies

In this section, we develop a series of case studies of increasing complexity both in terms of program size and also in terms of proof complexity. The examples we present are chosen not only to illustrate how Caliper is applied, but also to demonstrate how working in higher-order guarded separation logic allows for concise and composable specifications. For the sake of presentation, we make use of standard [Jung et al. 2018, §6] Hoare-triple notation

$$\{P\}\ e\ \{v.Q\} \triangleq \Box(P \mathrel{-\!\!*} \mathsf{rwp}\ e\ \{v.Q\})$$

throughout this section. The use of the *persistence* modality $\Box P$ says that P holds without asserting any exclusive ownership of resources and thus that it can be arbitrarily duplicated.

5.1 Recursion Through the Store

Recall the motivating example from the introduction, which uses Landin's knot to define a fixed-point combinator fix and then applies fix to define a recursive randomized program.

$$\mathsf{fix} \triangleq \lambda f.\ \mathsf{let}\ r = \mathsf{ref}\ (\lambda x.\ x)\ \mathsf{in}\ r \leftarrow (\lambda x.\ f\ (!\, r)\ x);\ !\, r$$

In our walk through of this example earlier in §3, we elided a discussion of how verification of the code making up fix itself works. In fact, as a first step, we may show a general higher-order specification for the fixed-point combinator fix. For all abstract predicates $\Phi, \Psi : \mathit{Val} \to \mathit{iProp}$, where iProp is the type of propositions in the logic, we show the specification

$$(\forall f, v'.\ \{\forall v''.\ \triangleright(\{\Phi(v'')\}\ f\ v''\ \{\Psi\})\}\ F\ f\ v'\ \{\Psi\} * \Phi(v')) \vdash \{\Phi(v)\}\ \mathsf{fix}\ F\ v\ \{\Psi\}.$$

The specification says that to prove postcondition Ψ of fix $F\ v$ given precondition $\Phi(v)$, it suffices to show a specification for F with postcondition Ψ. In proving the specification of F, however, one may assume that the first argument f (used for recursive calls) satisfies the specification as well, but under a later modality. Notice that the specification does not say anything explicitly about refinement—in fact, the same specification is given to fix in logics for partial correctness (see, *e.g.*, Birkedal and Bizjak [2023]) but *without* the later modality. In our specification, the later modality signifies an obligation to take a model step: intuitively, to recurse (and hence potentially not terminate), at least one step of the model must be exhibited in order for termination to be preserved. The proof of the specification is essentially identical to a proof carried out in standard Iris: after symbolically evaluating the store operation $!\,r$ one applies LÖB and the specification then follows by the assumed specification of F.

The fact that walk n refines the symmetric random walk model follows by showing the specification $\{\mathrm{spec}(n)\}$ walk n $\{\_.\exists m.\ \mathrm{spec}(m)\}$ which follows by applying the higher-order specification of fix, picking $\Phi(n) \triangleq \mathrm{spec}(n)$ and $\Psi(v) \triangleq \exists m.\ \mathrm{spec}(m)$. Instead of applying LÖB induction directly, as we did when initially describing the example, we apply the specification of f as provided by fix to recurse. A key benefit of this approach is that the use of a higher-order specification allows for a refinement proof that is agnostic to the intricacies of how iteration is realized. As a result, the proof can be re-used for different implementations of the recursor.

5.2 List Generators

Kobayashi et al. [2020] study probabilistic programs with higher-order functions (but without state) using probabilistic higher-order recursion schemes. As a motivating example, they present the following program that combines probabilistic choice and higher-order functions.

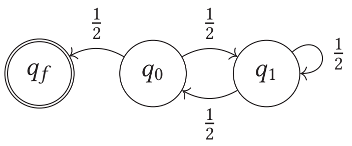
```
rec listgen f = if flip then None
                else let h = f () in
                     let t = listgen f in
                     Some (h, t)
```

The function listgen takes a generator f of elements as an argument and creates a list of elements, each of them obtained by calling f. The length of the list is randomized and distributed according to the geometric distribution. As a concrete application of listgen, they consider the program

$$\mathsf{listgen}\ (\lambda\_.\ \mathsf{listgen}\ (\lambda\_.\ \mathsf{flip}))$$

which generates a list of lists of random Booleans.

Using Caliper, we show that the program refines the model below.



Intuitively, state q_0 corresponds to the outer application of listgen and state q_1 to the inner application. Using the ranking super-martingale [Chakarov and Sankaranarayanan 2013] f that maps q_f, q_0, and q_1 to 0, 2, and 3, respectively, and $\epsilon = \frac{1}{2}$ one can straightforwardly show that the model almost-surely terminates.

First, we show a specification of the inner list generator

$$\{\mathrm{spec}(q_1)\}\ \mathsf{listgen}\ (\lambda\_.\ \mathsf{flip})\ \{\_.\,\mathrm{spec}(q_0)\}.$$

The proof proceed by LÖB induction. When we reach the flip expression in listgen, we apply RWP-COUPL-RAND using the coupling $\mathrm{step}(q_1) \lesssim \mathrm{unif}(\mathbb{B}) : (\lambda q, b.\ q = \text{if } b \text{ then } q_0 \text{ else } q_1)$. If b is true

```
init ≜ λ _. (ref None , tape 1)
cmp ≜ λℓ₁, ι₁, ℓ₂, ι₂. if ℓ₁ == ℓ₂ then 0 else cmpList ℓ₁ ι₁ ℓ₂ ι₂

rec cmpList ℓ₁ ι₁ ℓ₂ ι₂ =
  let (b₁, n₁) = getB ι₁ ℓ₁ in
  let (b₂, n₂) = getB ι₂ ℓ₂ in
  let c = cmpB b₁ b₂ in
  if c == 0 then cmpList n₁ ι₁ n₂ ι₂ else c

getB ≜ λι, ℓ. match !ℓ with
                 Some v ⇒ v
               | None   ⇒ let b = flip ι in
                          let n = ref None in
                          let v = (b, n) in
                          ℓ ← v;
                          v
               end

cmpB ≜ λb₁, b₂. if b₁ < b₂ then −1 else (if b₂ > b₁ then 1 else 0)
```

Fig. 3. Code for lazy uniform real sampling and comparison. Presampling annotations are shown in gray.

the goal is immediate. If b is false, we symbolically evaluate the flip expression in the generator using RWP-RAND as this second sampling is irrelevant to termination of the program. The induction hypothesis now finishes the proof.

The specification of the outer list generator looks as follows.

$$\{\text{spec}(q_0)\}\ \text{listgen}\ (\lambda\,\_.\ \text{listgen}\ (\lambda\,\_.\ \text{flip}))\ \{\_.\ \text{spec}(q_f)\}.$$

The proof proceeds by LÖB induction and we apply RWP-COUPL-RAND using the coupling $\text{step}(q_1) \lesssim \text{unif}(\mathbb{B}) : (\lambda q, b.\ q = \text{if } b \text{ then } q_f \text{ else } q_1)$. If b is true the goal is immediate. If b is false, the model is in state q_1 and we apply our specification for listgen $(\lambda\,\_.\ \text{flip})$ which returns the model in state q_0. The induction hypothesis finishes the proof.

While our methodology is sufficient for the example at hand, we would have liked to derive a single higher-order specification of listgen that suffices for proving both of the specifications above. However, to do so, we believe a richer notion of model is required. The function listgen keeps invoking f until it returns None, *i.e.*, until the corresponding stochastic process has terminated. But when listgen is invoked in a nested fashion, the process needs to be "restarted" for each nested invocation. To give a general, model-agnostic specification it seems that one would therefore need more model structure, *e.g.*, recursive Markov chains [Etessami and Yannakakis 2009].

5.3 Lazy Real

A standard result in probability theory says that sampling a real number uniformly from the interval $[0, 1]$ is equivalent to sampling an infinite sequence of Bernoulli random variables, each independently and uniformly drawn from the set $\{0, 1\}$ [Cohn 2013, Proposition 10.3.13]. We can think of the sequence of Bernoulli variables as representing the digits of the real number sampled from $[0, 1]$ written in binary form. Using this representation, we can implement a procedure to sample "exactly" from the uniform $[0, 1]$ distribution, by sampling these binary digits lazily as they are needed. In this example, we consider such a lazy implementation of a sampler, along with an operation for comparing the magnitude of two lazily-sampled reals.

An implementation is shown in Figure 3. The annotations shown in gray can be ignored for now and will be discussed later. We store the partially-sampled bits of a real number as a mutable linked list, where the head of the list is the most significant bit. The procedure init generates a fresh random sample with no bits sampled yet, as represented by a reference initialized to None. Then the comparison procedure cmp l_1 l_2 returns -1 if the real represented by l_1 is less than l_2, 0 if they are equal, and 1 if l_1 is greater than l_2. It is implemented by first checking whether the pointers l_1 and l_2 are equal. If they are, it short-circuits and immediately returns 0, since the corresponding real numbers must be the same. Otherwise, it calls cmpList, which recurses down the lists, checking bit by bit until it finds a position in the lists where the corresponding bits are not equal. Since the bits are only sampled lazily, during a call to cmpList, the next bit to be compared in one of the lists may not have been sampled yet. To handle this, cmpList uses the wrapper function getB when accessing a bit. The function getB either returns the bit (if it has already been sampled), and if not, it generates a fresh bit and appends it to the end of the linked list.

A priori, comparing two lazily-sampled reals with cmp is not guaranteed to terminate, as each generated bit of the two reals could be the same, indefinitely. However, cmp does terminate almost-surely. We will prove this by showing a refinement with the following Markov chain model, which samples from two independent coins until they disagree, as depicted by the following diagram.

$$\begin{tikzcd}
\boxed{\top\bot} & \boxed{\bot\top} \\
\top\top \arrow[u, bend left, "\frac{1}{4}"] \arrow[ur, "\frac{1}{4}"'] \arrow[r, bend left, "\frac{1}{4}"] \arrow[loop left, "\frac{1}{4}"] & \bot\bot \arrow[ul, "\frac{1}{4}"] \arrow[u, bend right, "\frac{1}{4}"'] \arrow[l, bend left, "\frac{1}{4}"] \arrow[loop right, "\frac{1}{4}"]
\end{tikzcd}$$

This model can be shown to almost-surely terminate using the ranking super-martingale $f(b_1, b_2) \triangleq$ if $b_1 \neq b_2$ then 0 else 2 with $\epsilon = 1$. In order to give a specification that supports multiple comparisons of (possibly) different lazily-sampled reals, we consider N iterations of the model above, *i.e.*, a model with state space $\mathbb{B} \times \mathbb{B} \times \mathbb{N}$ where the last component of the state tuple represents the remaining number of times we can call cmp.

As alluded to in §4, the main difficulties in showing the refinement are twofold: (1) when comparing random samples where fresh bits need to be sampled on both sides, one model step corresponds to two flip statements occurring in two different invocations of getB, and (2) when comparing random samples that have already had *some* bits sampled (but not the same amount), one of the samples might need to "catch up" by sampling additional bits first. Presampling tapes and asynchronous coupling are key ingredients in addressing both concerns in a high-level and composable manner. Thus, the first step in the proof is to consider a version of the program where the sampling operations are labeled with tapes. This version of the program is obtained by including the gray annotations shown in Figure 3.

To specify the lazily-sampled real operations we will make use of two predicates defined below.

$$Cmps(N) \triangleq \exists b, M.\, \mathsf{spec}(b, b, M) \land M \geq N$$

$$LazyReal(\vec{b}, v) \triangleq \exists \ell, \iota, \vec{b_1}, \vec{b_2}.\, v = (\ell, \iota) * \vec{b} = \vec{b_1} \cdot \vec{b_2} * IsList(\ell, \vec{b_1}) * \iota \hookrightarrow (1, \vec{b_2})$$

The $Cmps(N)$ predicate keeps track of the model resource and the fact that there are *at least* N comparison operations left. The $LazyReal(\vec{b}, v)$ predicate is a representation predicate that expresses that v corresponds to the lazy-sampled real denoted by the bits $\vec{b}$. Formally, the predicate says that v is a pair of a location ℓ and a tape label ι and $\vec{b}$ can be split into two sub-sequences $\vec{b_1}$ and $\vec{b_2}$ such that $\vec{b_1}$ corresponds to the linked list stored at location ℓ and $\vec{b_2}$ corresponds to bits that have been presampled onto the tape ι. The $IsList(\ell, l)$ assertion is a standard separation logic representation predicate for linked lists [Reynolds 2002].

We can now give general high-level specifications to the operations. When initializing a new lazily-sampled real, ownership of $LazyReal(\epsilon, v)$ is acquired for some v.

$$\{\mathsf{True}\}\ \mathsf{init}\ ()\ \{v.\, LazyReal(\epsilon, v)\}$$

When comparing two lazily-sampled reals, ownership of both reals are required as well as evidence that at least one comparison is left in the model.

$$\begin{array}{l}\left\{LazyReal(\vec{b_1}, v_1) * LazyReal(\vec{b_2}, v_2) * Cmps(N+1)\right\} \\ \quad \mathsf{cmp}\ v_1\ v_2 \\ \left\{\_.\, \exists \vec{b'_1}, \vec{b'_2}.\, LazyReal(\vec{b_1} +\!\!+ \vec{b'_1}, v_1) * LazyReal(\vec{b_2} +\!\!+ \vec{b'_2}, v_2) * Cmps(N)\right\}\end{array}$$

In the post condition we get back ownership of both reals, where more bits may have been sampled, and the number of comparisons has been decremented. Note that the return value of cmp is not important for establishing the refinement and can hence be ignored.

While the high-level specifications are intuitive, the proof of cmpList is more intricate and goes by induction on the (pre)sampled bit sequences $\vec{b_1}$ and $\vec{b_2}$. For the base case of the induction, which corresponds to reaching the end of both sequences, the proof uses Löb induction and presamples coupled bits to two tapes (which in turn allows us to eliminate the later modality). If both bit sequences are non-empty, the specification follows by symbolic execution and the induction hypothesis. If one sequence is empty and the other is not, we first sample additional bits using RWP-TAPE-EMPTY and continue as when both sequences are non-empty.

5.4 Treap

A *treap* [Seidel and Aragon 1996] is a randomized binary search tree structure. Rather than using rebalancing, it relies on randomness to ensure that the tree is $O(\log n)$ height with high probability. Searching for a key in the tree proceeds as normal in a binary search tree, but insertion makes use of randomness. To add a new key k into the tree, the insertion procedure first searches for k in the tree. If it finds k is already in the tree, insertion stops and returns. However, if k is not in the tree, insertion generates a random *priority* for k by sampling an element p independently from some totally ordered set. How these priorities are represented and the distribution on the set they are sampled from does not matter, so long as the probability of sampling the same priority twice is low. Once the priority p is generated, insertion creates a new node containing (k, p) and attaches it to the tree as a leaf node. At this point, the priority p is compared to the priority p' of the node's parent k'. If p is greater than p', then the insertion procedure performs a tree rotation, swapping the order of (k, p) and (k', p'). This rotation process is repeated recursively with the new parent of k, until k either has smaller priority than all of its ancestors, or it becomes the root.

As mentioned above, it is important for the priorities of all of the nodes to be distinct. If they are, then with high probability the tree will have $O(\log n)$ height. In theoretical analyses of treaps [Eberl et al. 2020; Seidel and Aragon 1996], it is common to treat the priorities as if they are real numbers sampled from some continuous distribution, so that the probability of a collision is 0.

Erickson [2017] notes that one may use lazily-sampled reals, as in the previous example, to represent the priorities. We show that with such an implementation of treaps, the insertions terminate almost surely. Of course, this follows from the fact that the comparison operation terminates almost surely, as the previous example showed. The motivation for this example is to demonstrate the modularity of our approach, as the treap proof does not need to know about the "internal" randomness and refinement proof of the lazy real comparisons.

Our specification makes use of a treap representation predicate $\mathit{IsTreap}(v, t)$ that expresses that v corresponds to the treap t as defined below.

$$\begin{aligned}\mathit{IsTreap}(v, \mathsf{leaf}) &\triangleq v = \mathsf{None}\\ \mathit{IsTreap}(v, \mathsf{node}(k, l, r)) &\triangleq \exists \ell, p, \vec{b}, v_l, v_r.\, v = \mathsf{Some}\ \ell * \ell \mapsto (k, p, v_l, v_r) *\\ &\qquad \mathit{LazyReal}(\vec{b}, p) * \mathit{IsTreap}(v_l, l) * \mathit{IsTreap}(v_r, r)\end{aligned}$$

If t is leaf, then $v = \mathsf{None}$. If t is $\mathsf{node}(k, l, r)$ then $v = \mathsf{Some}\ \ell$ and ℓ is a location that contain a tuple consisting of a key k, a priority p, and two treaps v_l and v_r. We omit the usual binary search tree invariants about the ordering of keys in l and r, since they are not needed to ensure termination. The $\mathit{LazyReal}(\vec{b}, p)$ assertion is the lazy real representation predicate from the previous section which states that p corresponds to the lazy-sampled real denoted by the bits $\vec{b}$. Notice how the structure of the definition closely resembles representation predicates for non-probabilistic data structures (such as lists, trees, *etc.*) and that the lazy real is logically managed abstractly through the $\mathit{LazyReal}(\vec{b}, p)$ assertion.

Given the representation predicate, our specification of the treap insert procedure looks as follow.

$$\begin{aligned}&\{\mathit{IsTreap}(v, t) * \mathit{Cmps}(N) * \ \mathit{height}(t) \leq N\}\\ &\quad \mathsf{insert}\ \ v\ k\\ &\{w.\mathit{IsTreap}(w, t') * \ \mathit{height}(t) \leq \mathit{height}(t') \leq \mathit{height}(t) + 1 * \mathit{Cmps}(M) * N - \mathit{height}(t) \leq M\}\end{aligned}$$

To insert a value k into the treap t, ownership of the treap is required and evidence that *at least* $\mathit{height}(t)$ comparisons are left in the lazy-real model. In return, we get ownership of a (possibly-) updated treap where the height may have been increased by one and $\mathit{height}(t)$ comparisons may have been performed. The proof proceeds in a straightforward way by applying the specification of cmp to compare the lazily-sampled priorities.

5.5 Galton-Watson Tree

In this example, we consider a sampler for generating Galton-Watson trees. Galton-Watson trees are random trees generated by the following stochastic process, which proceeds through a series of rounds. Initially, in round 1, the tree starts with a single root node, which we call generation 1 of the tree. In round 2, we sample a natural number n from some distribution μ, and attach n children nodes to the root node. These children are called generation 2. Inductively, in round $k + 1$, for each node i in generation k, we draw an independent sample n_i from μ and attach n_i children nodes to i. The nodes added in round $k + 1$ constitute generation $k + 1$. The process stops and is said to undergo *extinction* if a generation has no nodes, *i.e.* if all the n_i in a round are 0.

There are many algorithms for sampling Galton-Watson trees. One approach is to essentially follow the definition above for how the trees are generated. This can be seen as a kind of "breadth-first" sampling strategy, as we sample all the nodes in a given generation before moving on to any node in the next generation. In fact, one can consider alternate strategies for "traversing" the tree as it is generated. For example, Devroye [2012] describes a depth-first approach which maintains a stack containing the nodes whose children have not yet been sampled.[1]

Here, we consider an implementation of a Galton-Watson tree sampler that uses a stack to manage traversal of the tree. However, rather than storing *nodes* in the stack, we will use a higher-order

[1]Devroye in fact describes a variant where we want to sample a tree conditioned on the fact that it will have exactly k nodes for some k, so as a result the algorithm aborts and restarts if $k + 1$ nodes have been generated in a tree.

```
        rec sampleNode $d\ r\ s$ () =
          let $n = d$ () in
          let $f$ = ($\lambda$ _. let $r'$ = ref [] in (Stack.add (sampleNode $d\ r'\ s$) $s$); $r'$) in
          $r \leftarrow$ List.init $n\ f$

rec run $s$ = match Stack.take $s$ with          genTree $\triangleq \lambda d$. let $r$ = ref [] in
                Some $f \Rightarrow f$ (); run $s$                        let $s$ = Stack.create () in
              | None   $\Rightarrow$ ()                                    Stack.add (sampleNode $d\ r\ s$) $s$;
              end                                                 run $s$; ! $r$
```

Fig. 4. Implementation of a sampler for Galton-Watson trees in a higher-order event-loop style.

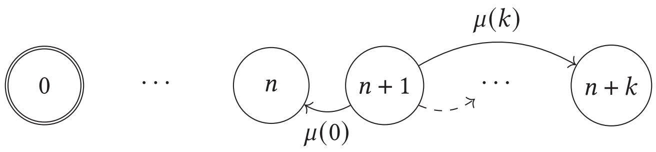
implementation that stores pending *tasks*, functions of type *unit* $\rightarrow$ *unit*, that when invoked will carry out sampling a given node's children. In addition, our implementation will be parameterized by a function d that carries out sampling from the distribution μ for the number of children.

The code is shown in Figure 4. A node is represented by a list of pointers to its children, with an empty list representing a leaf node. The top-level function genTree takes the child distribution sampling function d as an argument. It initializes a reference cell r that contains the root node, represented as an empty list (because r has no children yet) and creates an empty task stack s. An initial task sampleNode $d\ r\ s$ for sampling the children of r is added to the task stack. Then, the task stack run function is called, which invokes all the tasks in the stack until there are none remaining. When the call to run returns, genTree returns the representation of the root stored in r.

The actual work of carrying out sampling is done by the sampleNode task function, which takes as arguments the sampler function d, the node r to sample for, and the task stack s. This function begins by sampling the number of children n from d. It then uses the List.init function to initialize a list of length n, where each element of the list is a reference to a child node generated by a call to the locally defined function f. This function f adds a recursive task for the child node it generates to the task stack. The list of references is then stored back in r.

A key challenge here is that the task stack is re-entrant, in the sense that a task may add more tasks to the same stack it came from. Thus, in reasoning about the recursion in run, one cannot proceed by induction on the length of the stack, as the stack may grow before the recursive call.

When does such a sampler terminate? The sampler terminates only if the tree goes extinct. The probability of extinction is a classical problem in probability theory and depends on the distribution μ for the number of children. A typical proof approach is to represent the tree generation process as a random walk Markov chain, where the position of the random walk is the number of nodes that have not yet had their children sampled. Each transition of the walk corresponds to a node's children being sampled: if the node is in position $n + 1$, it transitions to $n + k$ with probability $\mu(k)$. (So in particular, moving from $n + 1$ to n if no children are produced.) The probability of extinction is equivalent to the probability that the walk hits 0. Graphically, we can represent this as follows:



We prove a specification that establishes a refinement between genTree and this Markov chain.

We first give general reusable specifications to the stack operations that do not concern themselves with refinement. Since the tasks in the stack-based sampler (*i.e.*, partial applications of sampleNode) add elements to the stack, we need a (self-referential) specification of the stack that allows the specification of the elements in the stack to depend on the stack itself.

By working in a logic with guarded recursion, we can define *guarded-recursive predicates* as a *guarded fixed point* $\mu x.\ t$. Guarded fixed points have no restriction on the variance of the recursive occurrence of x, as long as t is *contractive*, *i.e.*, as long as all the occurrences of x in t appear below a later modality. We will use such a predicate to define a self-referential representation predicate for the stack data structure to support the higher-order nature of the Galton-Watson sampler.

We define the predicate in two steps. First, we define an assertion $\mathit{IsStack}(s, n, \Phi)$ that expresses that s is a stack (implemented as a list) with n elements, each satisfying Φ.

$$\mathit{IsStack}(\Phi, s, n) \triangleq \exists l.\ \mathit{length}(l) = n * \mathit{IsList}(s, l) * \mathop{\Huge *}_{x \in l} \Phi(x)$$

Second, we define $\mathit{Stack}(\Psi)(s, n)$ as a guarded fixed point. The assertion expresses that s is a stack with n elements, each satisfying Ψ, but Ψ may also depend on the stack assertion.

$$\mathit{Stack}(\Psi)(s, n) \triangleq \mu Q.\ \mathit{IsStack}((\lambda w.\ \Psi(Q, w, s)), s, n)$$

For the fixed point to exist, the predicate $\Psi : (\mathit{Val} \rightarrow \mathbb{N} \rightarrow \mathit{iProp}) \rightarrow \mathit{Val} \rightarrow \mathit{Val} \rightarrow \mathit{iProp}$ must be contractive in the first parameter.

With the $\mathit{Stack}(\Psi)$ representation predicate in hand, we prove the following generic specifications for the stack operations.

$\{\mathsf{True}\}\ \mathsf{Stack.create}\ \{s.\ \mathit{Stack}(\Psi)(s, 0)\}$

$\{\mathit{Stack}(\Psi)(s, n) * \Psi(\mathit{Stack}(\Psi), v, s)\}\ \mathsf{Stack.add}\ v\ s\ \{\_.\ \mathit{Stack}(\Psi)(s, n + 1)\}$

$\{\mathit{Stack}(\Psi)(s, 0)\}\ \mathsf{Stack.take}\ q\ \{v.\ v = \mathsf{None}\ *\ \mathit{Stack}(\Psi)(s, 0)\}$

$\{\mathit{Stack}(\Psi)(s, n + 1)\}\ \mathsf{Stack.take}\ s\ \{v.\ \exists w.\ v = \mathsf{Some}\ \ w *\ \mathit{Stack}(\Psi)(s, n) * \Psi(\mathit{Stack}(\Psi), w, s)\}$

When creating a stack, ownership of an empty stack $\mathit{Stack}(\Psi)(s, 0)$ is obtained for a user-chosen contractive stack predicate Ψ. To add an element v to the stack s, ownership of the stack and $\Psi(\mathit{Stack}(\Psi), v, s)$ is required. Finally, when popping an element v from a non-empty stack, one gets back ownership of $\Psi(\mathit{Stack}(\Psi), v, s)$.

To specify the Galton-Watson sampler, we apply our general specification of the stack operations. Recall that genTree stores suspended tasks, *i.e. closures*, in the task stack. Intuitively, this means that the stack predicate Ψ_{GW} we instantiate the stack library with must be a *specification* of the shape $\triangleright (\{\mathit{Stack}(\Psi)(s, n) * \ \ldots\}\ f\ ()\ \{\_.\ \mathit{Stack}(\Psi)(s, n + k) * \ \ldots\})$ for task f. The Hoare triple must be behind a later modality for Ψ_{GW} to be contractive since the pre- and postcondition contains the stack representation predicate. This may seem like an obstacle to specifying the run event-loop: when suspended tasks f are taken out of the stack, the specification will still appear below a later modality which the proof has no way of eliminating. That is, we will need to eliminate the later modality *now* to apply the specification of f, but a coupling step—which would allow us to eliminate the later modality—only happens *in the future* when f is invoked. To address this issue, the key idea is to *asynchronously* couple the sampling happening in d with the model during the proof of the run specification and thus eliminate the later modality earlier than otherwise permitted. As our language and thus presampling tapes only support uniform sampling, we assume μ distributes as $\mathsf{unif}(N)$ for some N for the remainder of the section.

Our specification of genTree looks as follows.

$$\{\iota_{\mathrm{GW}} \hookrightarrow (N,k)\}\ d\ ()\ \{v.\, v = k * \iota_{\mathrm{GW}} \hookrightarrow (N,\epsilon)\} \vdash$$
$$\{\mathsf{spec}(1) * \iota_{\mathrm{GW}} \hookrightarrow (N,\epsilon)\}\ \mathsf{genTree}\ d\ \{\mathsf{True}\}$$

The specification requires that the sampling function d consumes randomness from the ι_{GW} tape when invoked; the run specification will populate the presampling tape through asynchronous coupling. When creating the stack, we pick the stack predicate Ψ_{GW} below.

$$\Psi_{\mathrm{GW}}(S,s,f) \triangleq \triangleright(\forall n,k.\ \{S(s,n) * \iota_{\mathrm{GW}} \hookrightarrow (N,k)\}\ f\ ()\ \{\_.\, S(s,n+k) * \iota_{\mathrm{GW}} \hookrightarrow (N,\epsilon)\})$$

The predicate says that tasks in the stack must satisfy a specification that, given ownership of the stack and $\iota_{\mathrm{GW}} \hookrightarrow (N,k)$, adds k new tasks to the stack.

The crux of the proof lies in the specification of run.

$$\{\mathit{Stack}(\Psi_{\mathrm{GW}})(s,n) * \mathsf{spec}(n) * \iota_{\mathrm{GW}} \hookrightarrow (N,\epsilon)\}$$
$$\mathsf{run}\ s$$
$$\{\_.\, \exists m.\ \mathit{Stack}(\Psi_{\mathrm{GW}})(s,m) * \mathsf{spec}(m) * \iota_{\mathrm{GW}} \hookrightarrow (N,\epsilon)\}$$

The specification of run requires ownership of a stack s with n elements, the model resource at state n, and the ι_{GW} presampling tape. The proof also proceeds by Löb induction. To eliminate *both* the later modality in front of the induction hypothesis *and* the later modality in front of the specification of the task retrieved from the queue, the proof asynchronously couples the model transition from n with a presampling onto ι_{GW}.

6 Semantic Model and Soundness

The soundness of Caliper is justified by constructing a semantic model of $\mathsf{rwp}\ e\ \{\Phi\}$ in terms of the Iris "base logic" [Jung et al. 2018]. This underlying base logic is a separation logic with the various connectives we saw in earlier sections, including the $\triangleright$ modality and the Löb induction principle. This section defines the semantic model and then shows how the model implies Theorem 3.1. For notational convenience, we write an inference rule with premises $P_1, \ldots, P_n$ and conclusion Q as notation for $(P_1 * \ldots * P_n) \vdash Q$ in the Iris base logic.

6.1 Model

The semantic model of the refinement weakest precondition $\mathsf{rwp}\ e\ \{\Phi\}$ constructs a coupling of the execution of m as tracked by the $\mathsf{spec}(m)$ resource and the execution of the program e. Intuitively, it does so by constructing individual stepwise couplings as the proof symbolically executes the program and the model. In the end, these stepwise couplings will all be combined to construct a coupling of the full executions.

In contrast to many logics making use of guarded recursion, $\mathsf{rwp}\ e\ \{\Phi\}$ is defined as a *least* fixed point. This is reminiscent of models for weakest preconditions that ensure total program correctness but our simultaneous use of guarded recursion will permit non-termination in a controlled way. Formally, the least fixed point $\mathrm{lfp}\, x.t$ exists if t is monotone, *i.e.*, all recursive occurrences of x appear in a positive position, as follows from Tarski's fixed-point theorem [Tarski 1955]. The definition looks as follows.

$$\begin{aligned}\mathsf{rwp}\ e_1\ \{\Phi\} \triangleq\ \mathrm{lfp}\, W.&(e_1 \in \mathit{Val} * \Phi(e_1)) \lor \\ &\quad (e_1 \notin \mathit{Val} * \forall m_1, \sigma_1.\ M(m_1) * S(\sigma_1) \mathrel{-\!\!*} \\ &\qquad \mathsf{cpl}\ m_1 \sim (e_1,\sigma_1)\ \{m_2, (e_2,\sigma_2).\ M(m_2) * S(\sigma_2) * W(e_2,\Phi)\})\end{aligned}$$

The left disjunct of the definition says that if e_1 is a value, then the postcondition $\Phi(e_1)$ must hold. Meanwhile, the right side says if e_1 is *not* a value, then we get to assume ownership of two resources

$$\frac{\text{reducible}(\rho_1) \qquad \text{ret}(m) \lesssim \text{step}(\rho_1) : R \qquad \forall s \in R.\, \Psi(s)}{\text{cpl}\, m \sim \rho_1\, \{\Psi\}}\ \textsc{cpl-prog}$$

$$\frac{\text{reducible}(\rho_1) \qquad \text{reducible}(m_1) \qquad \text{step}(m_1) \lesssim \text{step}(\rho_1) : R \qquad \forall s \in R.\, \triangleright \Psi(s)}{\text{cpl}\, m_1 \sim \rho_1\, \{\Psi\}}\ \textsc{cpl-model-prog}$$

$$\frac{\text{reducible}(m_1) \qquad \text{step}(m_1) \lesssim \text{ret}(\rho) : R \qquad \forall (m_2, \rho) \in R.\, \triangleright\, \text{cpl}\, m_2 \sim \rho\, \{\Psi\}}{\text{cpl}\, m_1 \sim \rho\, \{\Psi\}}\ \textsc{cpl-model}$$

Fig. 5. Inductive definition of the coupling precondition cpl $m \sim \rho\ \{\Psi\}$.

$M(m_1)$ and $S(\sigma_1)$ and have to prove a *coupling precondition* cpl $m_1 \sim (e_1, \sigma_1)\ \{\ldots\}$ as defined below. The postcondition of the coupling precondition requires the prover to give back the two updated resources and show that rwp $e_2\ \{\Phi\}$ holds recursively.

The two resources $M(m_1)$ and $S(\sigma_1)$ are, respectively, a model and a state *interpretation*. Formally, they track *authoritative* views of the model and the state [Jung et al. 2015]. The model interpretation always agrees with the model state tracked with the spec(m) resource, *i.e.*, $M(m) * \text{spec}(m') \vdash m = m'$, and the state interpretation always agrees with the points-to connective $\ell \mapsto v$ for the heap, *i.e.*, $S(\sigma) * \ell \mapsto v \vdash \sigma(\ell) = v$.

The coupling precondition is the heart of the probabilistic program logic and ensures (1) the *safety* of (e_1, σ_1), meaning that it does not get stuck, and (2) the existence of a relational coupling with the model. The connective cpl $m \sim \rho\ \{\Psi\}$ is a ternary relation on a model state $m \in M$, a program configuration $\rho \in \mathit{Cfg}$, and a relational post condition $\Psi : M \to \mathit{Cfg} \to \mathit{iProp}$ where *iProp* is the type of propositions in the logic. Intuitively, it forms a relational coupling logic that establishes the existence of a probabilistic coupling of *one* step of the program configuration ρ with a number of steps of the model m such that the postcondition Ψ holds for the support. Formally, it is defined inductively by the inference rules shown in Figure 5 (*i.e.* as a least fixed point).

The first constructor cpl-prog applies to symbolic steps that only progress the program. It requires the configuration to be reducible and that there is a trivial coupling between the Dirac distribution of the model state ret(m) and the program step, which just means that the program configuration can take a step. Moreover, everything in the support of the coupling must satisfy the postcondition. The constructor is essential to validating all the unary program logic rules shown in Figure 1.

The second constructor cpl-model-prog is used to validate rwp-coupl-rand, the coupling rule in the logic. It requires that the model and the configuration are reducible (to guarantee safety) and that a coupling can be exhibited between a step of the model and a step of the program. Finally, everything in the support of the coupling must satisfy Ψ but *under a later modality*. This occurrence is one of the two places that formally connects the later modality to steps of the model.

The third and last constructor cpl-model is used to validate rwp-spec-det, and thus the symbolic steps which only progress the model. It requires a trivial coupling of the Dirac distribution of the program configuration with the model step, which intuitively just means that the model can take a step. For everything in the support of the coupling, the coupling precondition must hold recursively *under a later modality*, again connecting the later modality to steps of the model.

The definition of the refinement weakest precondition consists of multiple interacting components: probabilistic couplings, later modalities, resources, and two fixed points. It is this fine (and

$$\textsc{ref-val}\quad \frac{v \in \mathit{Val}}{m \precsim (v, \sigma)} \qquad \textsc{ref-prog}\quad \frac{\text{reducible}(\rho_1) \qquad \text{ret}(m) \precsim \text{step}(\rho_1) : R \qquad \forall (m, \rho_2) \in R.\, m \precsim \rho_2}{m \precsim \rho_1}$$

$$\textsc{ref-model-prog}\quad \frac{\text{reducible}(\rho_1) \qquad \text{reducible}(m_1) \qquad \text{step}(m_1) \precsim \text{step}(\rho_2) : R \qquad \forall (m_2, \rho_2) \in R.\, \triangleright m_2 \precsim \rho_2}{m_1 \precsim \rho_1}$$

$$\textsc{ref-model}\quad \frac{\text{reducible}(m_1) \qquad \text{step}(m_1) \precsim \text{ret}(\rho) : R \qquad \forall (m_2, \rho) \in R.\, \triangleright m_2 \precsim \rho}{m_1 \precsim \rho}$$

Fig. 6. Inductive definition of the (plain) guarded refinement relation $m \precsim \rho$.

subtle!) balance of the components that allows us in the following section to prove that termination is indeed preserved across the program and the model, but it is also what allows us to enable the reasoning principles that we want. For example, the fact that *one* unfolding of the rwp $e\ \{\Phi\}$ fixed point always corresponds to *one* step of the program (but possibly multiple steps of the model) is crucial to stating and proving soundness of the rules in Figure 1.

6.2 Soundness

We show soundness of Caliper in two stages:

(1) we show that the relational logic establishes a so-called "plain" guarded refinement $m \precsim \rho$, *i.e.* a guarded relation that does not depend on separation logic resources, and
(2) we show that plain guarded refinement implies preservation of termination.

The plain guarded refinement is defined inductively by the rules in Figure 6. If the program has terminated the refinement trivially holds (ref-val), we can step the program and the model independently (ref-prog and ref-model, respectively), and we can incorporate non-trivial couplings of model and program steps (ref-model-prog). In the two cases where we progress the model (ref-model and ref-model-prog), the recursive occurrence of the refinement is under a later modality, thus connecting the later modality to steps of the model as in the definition of the relational logic.

The first stage of the soundness proof is the following lemma.

Lemma 6.1. *If* $\text{spec}(m) \vdash \text{rwp}\ e\ \{\Phi\}$ *then* $\vdash m \precsim (e, \sigma)$ *for all* σ.

The proof goes by structural induction in both the weakest precondition rwp $e\ \{\Phi\}$ and the coupling precondition cpl $m \sim \rho\ \{\Psi\}$ fixed points. While the details of how resources are erased depends on how they are managed in Iris (*e.g.*, through the fancy update modality, which we have omitted), for intuition about why this should hold, observe that *if* we ignore the model and state interpretation resources, each case of rwp $e\ \{\Phi\}$ and the constructors of cpl $m \sim \rho\ \{\Psi\}$ correspond exactly to one constructor of the $m \precsim \rho$ refinement relation.

The core of the soundness proof is the second stage and the fact that the $m \precsim \rho$ relation preserves termination. The key enabler is the monotone convergence of termination probability (Lemma 2.5), that is, to show that the termination probability of the program is bounded below by the termination probability of the model, it suffices to consider all finite prefixes of the model execution—exactly what our guarded refinement relation is concerned with. This in turn means that to show that termination is preserved, we "just" have to combine the stepwise left-partial couplings constructed in the refinement relation into a single coupling of executions. The ability to combine couplings in this way follows from the following lemma showing that left-partial couplings can be composed along the monadic structure of sub-distributions:

Lemma 6.2 (Composition of couplings)**.** *Let $R \subseteq A \times B$, $S \subseteq A' \times B'$, $\mu_1 \in \mathcal{D}(A)$, $\mu_2 \in \mathcal{D}(B)$, $f_1 : A \to \mathcal{D}(A')$, and $f_2 : B \to \mathcal{D}(B')$.*

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

Using this lemma, we have the following:

Lemma 6.3. *$m \lesssim \rho \vdash \triangleright^n \mathrm{exec}_n(m) \lesssim \mathrm{exec}(\rho)$ for all n.*

Proof. The proof proceeds by induction on the $m \lesssim \rho$ fixed point.

Case ref-val**.** Since $\rho = (v, \sigma)$ and $v \in \mathit{Val}$ we get that $\mathrm{exec}(v, \sigma) = \mathrm{ret}(v)$. As $\mu \lesssim \mathrm{ret}(v)$ trivially holds for any μ (pick a coupling that relates all the mass of μ to v), we conclude.

Case ref-prog**.** Since ρ is reducible, we get that $\mathrm{exec}(\rho) = \mathrm{step}(\rho) \gg\!= \mathrm{exec}$. By the left identity law of the distribution monad, $\mathrm{exec}_n(m) = \mathrm{ret}(m) \gg\!= \mathrm{exec}_n$. We are left with the goal

$$\triangleright^n(\mathrm{ret}(m) \gg\!= \mathrm{exec}_n \lesssim \mathrm{step}(\rho) \gg\!= \mathrm{exec}).$$

Using later-mono we apply Lemma 6.2 under the later modalities and exploit the coupling $\mathrm{ret}(m) \lesssim \mathrm{step}(\rho_1) : R$ which leaves us with the goal

$$\triangleright^n\, \forall (m, \rho_2) \in R.\ \mathrm{exec}_n(m) \lesssim \mathrm{exec}(\rho_2)$$

which follows by the induction hypothesis.

Case ref-model-prog**.** We do a case distinction on n. If $n = 0$ then $\mathrm{exec}_0(m) = \mathbf{0}$ and thus the left-partial coupling exists trivially. If $n \neq 0$ then $\mathrm{exec}_n(m) = \mathrm{step}(m) \gg\!= \mathrm{exec}_{n-1}$ and since ρ is reducible, we get that $\mathrm{exec}(\rho) = \mathrm{step}(\rho) \gg\!= \mathrm{exec}$. This leaves us with the goal

$$\triangleright^n(\mathrm{step}(m) \gg\!= \mathrm{exec}_{n-1} \lesssim \mathrm{step}(\rho) \gg\!= \mathrm{exec})$$

which follows as above by later-mono, Lemma 6.2, and the induction hypothesis.

Case ref-model**.** We do a case distinction on n. If $n = 0$ then $\mathrm{exec}_0(m) = \mathbf{0}$ and thus the left-partial coupling exists trivially. If $n \neq 0$ then $\mathrm{exec}_n(m) = \mathrm{step}(m) \gg\!= \mathrm{exec}_{n-1}$ and by the left identity law $\mathrm{exec}(\rho) = \mathrm{ret}(\rho) \gg\!= \mathrm{exec}$. This leaves us with the goal

$$\triangleright^n(\mathrm{step}(m) \gg\!= \mathrm{exec}_{n-1} \lesssim \mathrm{ret}(\rho) \gg\!= \mathrm{exec})$$

which follows as above by later-mono, Lemma 6.2, and the induction hypothesis. □

The result shows that the desired coupling exists, but this existence is *internal* to the Iris base logic, and under n iterations of the $\triangleright$ modality. At this point we rely on the following soundness theorem for the Iris base logic to know that the coupling exists externally in the meta-logic:

Theorem 6.4. *Let φ be a meta-logic proposition. If $\vdash \triangleright^n \varphi$ then φ holds in the meta-logic.*

Corollary 6.5. *If $\vdash m \lesssim \rho$ then $\mathrm{exec}_n(m) \lesssim \mathrm{exec}(\rho)$ for all n.*

Proof. Immediate by applying Theorem 6.4 and Lemma 6.3. □

Corollary 6.6. *If $\mathrm{spec}(m) \vdash \mathrm{rwp}\ e\ \{\Phi\}$ then $\mathrm{exec}_n(m) \lesssim \mathrm{exec}(e, \sigma)$ for all n and σ.*

Proof. Immediate by applying Corollary 6.5 and Lemma 6.1. □

The soundness theorem of Caliper (Theorem 3.1) then follows directly by applying Lemma 2.5, Lemma 2.8, and Corollary 6.6.

Asynchronous couplings. To incorporate asynchronous couplings into Caliper, we add a fourth constructor to the coupling precondition that adds the possibility of coupling a model step with any number of presampling steps. The distribution foldM(sstep, σ_1, l) denotes a monadic fold of sstep over the list l of tape labels using σ_1 as the initial value.

$$\frac{l \subseteq \mathrm{dom}(\sigma_1) \qquad \begin{array}{c}\mathsf{reducible}(m_1)\\ \mathsf{step}(m_1) \lesssim \mathrm{foldM}(\mathsf{sstep}, \sigma_1, l) : R\end{array} \qquad \forall (m_2, \sigma_2) \in R.\ \triangleright \mathsf{cpl}\ m_2 \sim (e_1, \sigma_2)\ \{\Psi\}}{\mathsf{cpl}\ (e_1, \sigma_1) \sim m_1\ \{\Psi\}}$$

The state interpretation is extended accordingly to give meaning to the $\iota \hookrightarrow (N, \vec{n})$ resource.

As for the coupling precondition we also extend the plain guarded refinement relation in Figure 6. The soundness theorem can then adapted by making use of Lemma 4.1 to erase presampling steps.

6.3 Comparison to Guarded Recursion for Non-Probabilistic Termination Preservation

As mentioned earlier, prior works have also built program logics for termination-preserving refinements using guarded recursion [Spies et al. 2021a; Tassarotti et al. 2017; Timany et al. 2024a]. These works target non-probabilistic languages that instead have (adversarial) non-determinism.

In terms of logical rules, these works use similar core mechanisms of (1) representing a specification program or model as ghost state, (2) reasoning about loops using Löb induction, and (3) only allowing the later modality to be eliminated when the specification program takes a step. Caliper differs primarily in that its coupling rule requires the resolution of probabilistic non-determinism between the program and the model to have corresponding probabilities, whereas in these prior works, non-determinism at the model level is resolved angelically. Of course, the resulting soundness theorems for these prior logics also differ. They say that if the program has a non-terminating execution, then the model must also have a non-terminating execution.[2]

The other key difference is that to prove their soundness theorems, these prior works have found it necessary to either move to transfinite step indexing [Svendsen et al. 2016], or to require some kind of *finiteness* condition, *e.g.* require that models be *finitely branching* (meaning that each state can only move to finitely many states in a single transition) or relative image-finiteness of the refinement relation. Caliper's soundness proof requires no such restriction. The model uses "standard" natural number step indexing, and a Markov chain model is allowed to have countable branching, since Definition 2.4 permits the support of the chain's step function to be countable.

One might wonder why these technical workarounds were not needed in the probabilistic case, and whether Caliper's soundness proof implies that something similar could be done in the non-deterministic case as well. The answer lies in Lemma 2.5, which shows that the termination probability of a program can be lower-bounded by considering the termination probability of its n-step finite approximations. This theorem was used in the soundness proof of Caliper, allowing us in Corollary 6.6 to consider executions of up to n steps of the model for each n. Only considering n-step executions was important because these corresponded to the up to n iterations of the later modality incurred when unfolding the guarded recursion defining the coupling precondition.

A corresponding proof approach does not work in the context of non-probabilistic termination, because there is no useful analogue of Lemma 2.5. In general, knowing that for all n, a non-deterministic program has an execution that has not terminated after n steps does not imply that it necessarily has a diverging execution. Thus, the soundness proofs in the aforementioned logics cannot use the approach of considering n step unfoldings of guarded recursion that we used above.

[2] Tassarotti et al. [2017] and Timany et al. [2024a] consider a concurrent language and allow for a stronger property, requiring that if the non-terminating execution in the program was under a *fair* scheduler, then the execution in the model must also be fair. This imposes some restrictions on how non-determinism at the model is resolved, so that it is not entirely angelic.

7 Related Work

AST of first-order programs. The study of termination of stochastic processes has a long history in probability theory, and there are broad classes of techniques for proving termination in the theory of Markov chains and branching processes. Many of these techniques have been adapted to formal methods for proving termination of probabilistic programs.

For example, Chakarov and Sankaranarayanan [2013] use ranking super-martingales (RSM) for proving almost-sure termination. They implement an analysis that can automatically discover the existence of RSMs. A number of follow-up works have extended the scope and applicability of ranking super-martingales [Agrawal et al. 2018; Fioriti and Hermanns 2015; Fu and Chatterjee 2019; McIver et al. 2018]. Several of these works develop program logics for first-order imperative probabilistic programs, in which the primitive mechanism for looping is a `while` loop construct. Ranking super-martingales are used in the conditions for the `while` loop, analogous to the way ranking functions are used in standard, non-probabilistic Hoare logic's variant rule for loops. In principle, similar rules could be devised for particular schemes and patterns of recursion in a higher-order language like ProbLang, but it would be challenging to devise general purpose rules, that would apply to examples like the treap or the Galton-Watson sampler from §5.

Arons et al. [2003] give an approach for reducing almost-sure termination of a randomized program to may-termination under a non-deterministic semantics. For this non-deterministic semantics, their *planner* rule allows one to pre-select a finite sequence of outcomes for random choices. If the program can be shown to terminate when this finite sequence of outcomes occurs, then it must terminate almost-surely under the standard probabilistic semantics. Esparza et al. [2012] generalize this rule to more flexible forms of finite sequences, which they call *patterns*. This generalization is complete for *weakly finite* programs, which from each starting state can only reach a finite number of states. An advantage of this approach is that it allows one to then apply tools and algorithms for automatically analyzing non-probabilistic program termination. However, examples like the unbounded random walk or the Galton-Watson tree are not weakly finite in this sense, and appear to be beyond the scope of these techniques. The soundness of these approaches is justified by the Borel-Cantelli lemma, which shows that the pre-selected sequence of outcomes must occur infinitely often in any non-terminating execution. The Borel-Cantelli lemma is an example of a zero-one law, and similar zero-one laws have been used to justify other proof rules for almost sure termination [Hart et al. 1983].

Positive almost-sure termination (PAST) is a stronger property [Bournez and Garnier 2005; Majumdar and Sathiyanarayana 2024], stating that the expected number of total transitions for a program is finite. A number of methods have been developed for proving bounds on the expected running time of a program, thereby implying almost-sure termination. For example, Kaminski et al. [2016] develop a weakest-precondition style calculus for bounding expected running time. Ngo et al. [2018] develop an extension of automatic amortized resource analysis (AARA) for bounding expectations of resource use in randomized programs. Some of the examples considered in §5, such as the random walk, are AST but have infinite expected running time, so techniques based on bounding expected running time cannot be used to prove that they are AST.

Although our focus has been on using Caliper to prove AST, the lower bound on termination probabilities established by Theorem 3.1 applies even when the model does not terminate with probability 1. Thus, Caliper can be used to show lower-bounds on termination probabilities for programs that are not AST. Feng et al. [2023] develop a weakest pre-expectation calculus for proving lower bounds on expected values of quantities in non-AST programs.

AST of higher-order programs. Hurd [2002] develops theory for proving termination of a monadic embedding of randomized programs in HOL. He defines a `while` combinator for this embedding and proves analogues of 0-1 rules for almost-sure termination developed by Hart et al. [1983].

Avanzini et al. [2021] present a technique to reason about the expected runtime of programs written in a probabilistic lambda calculus. They use a continuation-passing style translation, where the continuation maps inputs to expected runtimes, thus turning the program into a cost transformer.

Several works introduce type systems that imply termination or expected bounds on higher-order programs. Lago and Grellois [2017] present a probabilistic variant of sized types that ensures almost-sure termination of well-typed programs. In this system, a function body's type is associated with a particular kind of random walk called a sized random walk. The typing rule for `letrec` has a premise that requires the corresponding random walk to be almost-surely terminating. AST for these sized random walks is shown to be decidable. Lago et al. [2021] develop an intersection type systems for capturing both almost-sure and positive almost-sure termination. Wang et al. [2020] present a probabilistic variant of RaML, a higher-order language with a type system that does amortized resource analysis to produce bounds on expected resource consumption automatically.

Kobayashi et al. [2020] introduce probabilistic higher-order recursion schemes (PHORS), a probabilistic variant of the HORS considered in higher-order model checking. They show that the decision problem for almost-sure termination of order-2 PHORS is undecidable, and introduce a sound but incomplete procedure for bounding termination probabilities of order-2 PHORS. An implementation of their procedure is applied to several PHORS, including one that is equivalent to the listgen example in §5. In place of Markov chains, PHORS could be used as the models in Caliper, in order to prove that a particular PHORS is an adequate abstraction of a program.

In contrast to Caliper, the above works deal with higher-order calculi *without* imperative state.

Guarded recursion for termination and termination-preserving refinement. We have already discussed closely related uses of guarded recursion for termination-preserving refinement in §6.3. Other works have made use of guarded recursion or step-indexed models to prove termination or termination-preserving refinement. Spies et al. [2021b] construct a transfinite step-indexed logical relation to show that a linear type system implies termination. SeLoC [Frumin et al. 2021a] is a relational logic for proving termination-sensitive noninterference properties of higher-order programs. To achieve the intended security property, the simulation relation that their program logic encodes is much stricter than the one we have considered here, so that later modalities are only eliminated in rules where *both* the program and its model take a simultaneous step. In contrast, Gregersen et al. [2021] develop a model of termination-*in*sensitive noninterference where later modalities are eliminated when *either* the program or the model take a step.

Much as PAST implies AST, establishing an upper bound on the number of steps a program takes also implies termination. Several works have proved resource bounds in step-indexed program logics [Mével et al. 2019; Pottier et al. 2024] using the technique of time credits [Atkey 2011; Charguéraud and Pottier 2019]. In this approach, a later modality is eliminated on every program step, which might at first appear to allow infinite looping by Löb induction. However, this is ruled out by requiring a finite resource called a time credit to be spent on steps of execution. Such an approach could be adapted to deriving expected bounds on probabilistic programs, but this would not be applicable for proving AST of the symmetric random walk, as it is not PAST.

Guarded recursion for probabilistic refinement. Guarded recursion has also been used to define logical relations models and logics for refinement of probabilistic programs. Bizjak and Birkedal [2015] construct such a logical relation for a language similar to our ProbLang, but without higher-order state. Aguirre and Birkedal [2023] extend this language with support for countable non-deterministic choice. In the presence of non-determinism, a program has a range of probabilities of termination based on how non-determinism is resolved by a scheduler. They develop a logical relation for proving equivalences with respect to the may and must-termination probabilities (the maximal and minimal probabilities across all schedulers). Wand et al. [2018] build a step-indexed logical relation for proving contextual equivalences for a higher-order language with sampling from continuous distributions. Finally, Clutch [Gregersen et al. 2024b] is a program logic based on Iris for proving contextual refinements (that do not preserve termination of programs) written in a language like ProbLang. The general structure of our coupling precondition is inspired by the coupling modality of Clutch. However, our definitions, method for preserving termination, and the soundness proof is new and entirely different. We re-use the idea of pre-sampling tapes from Clutch, which is mostly orthogonal to the question of termination preservation.

In the above works, the defined refinement relations lead to lower bounds that are effectively in the *opposite* direction of Theorem 3.1. In other words, translating their approaches to our setting leads to a soundness theorem in which the termination probability of the *program* is a lower bound on the termination probability of the *model*. This is because their approaches effectively allow for later elimination when the program takes a step, as opposed to when the model takes a step as in Caliper.[3] Since the goal in those works is to prove *equivalences* by proving refinements in both directions, the direction of this inequality is adequate for their purposes.

Aguirre et al. [2018] present a logic to prove couplings between pairs of infinite runs over Markov chains defined in a probabilistic guarded lambda calculus, without considering any form of state. Markov chains are defined as distributions over infinite streams. Productivity (the fact that a step will eventually be taken) is dual to termination, and is ensured by their type system.

Polaris [Tassarotti and Harper 2019] is a concurrent program logic based on Iris for proving a coupling between a randomized program and a more abstract model. The soundness theorem for Polaris allows bounds on probabilities and expectations in the model to be translated into bounds on the program across schedulers. However, these bounds only apply under schedulers for which the program terminates in a bounded number of steps. Thus, Polaris is a kind of partial correctness logic, as its soundness theorem *assumes* a property that is already stronger than AST.

8 Conclusion

We have presented Caliper, a logic for proving termination-preserving refinements between higher-order probabilistic programs and more abstract models. For proving such refinements, Caliper combines powerful techniques such as Löb induction and couplings. We have demonstrated Caliper on several examples, including ones that are outside the scope of prior methods and approaches for proving almost-sure termination of randomized programs. A natural future direction would be to extend Caliper with support for adversarial non-determinism in programs and models, and to consider classes of abstract models with more structure, such as PHORS and recursive Markov chains, with the aim of developing more composable specifications.

Acknowledgments

This work was supported in part by the National Science Foundation, grant no. 2225441, the Carlsberg Foundation, grant no. CF23-0791, a Villum Investigator grant, no. 25804, Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation, and the European Union (ERC, CHORDS, 101096090). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

[3]The first three of these works use explicitly step-indexed models, as opposed to the "logical" approach to step indexing [Dreyer et al. 2011] with $\triangleright$ modalities. Nevertheless, they decrement the step index when the program on the left side of their refinement relation takes a step, which corresponds to eliminating a later modality as we have described.

A Refinement Logic with Invariant Mask Annotations

Lemma A.1 (Program logic rules).

$$e_1 \rightsquigarrow e_2 * \mathsf{rwp}_{\mathcal{E}}\, e_2\, \{\Phi\} \vdash \mathsf{rwp}_{\mathcal{E}}\, e_1\, \{\Phi\} \qquad \text{RWP-PURE}$$

$$\forall \ell.\, \ell \mapsto v \mathrel{-\!\!*} \Phi(\ell) \vdash \mathsf{rwp}_{\mathcal{E}}\, \mathsf{ref}\, v\, \{\Phi\} \qquad \text{RWP-ALLOC}$$

$$(\ell \mapsto v \mathrel{-\!\!*} \Phi(v)) * \ell \mapsto v \vdash \mathsf{rwp}_{\mathcal{E}}\, !\,\ell\, \{\Phi\} \qquad \text{RWP-LOAD}$$

$$(\ell \mapsto w \mathrel{-\!\!*} \Phi(())) * \ell \mapsto v \vdash \mathsf{rwp}_{\mathcal{E}}\, \ell \leftarrow w\, \{\Phi\} \qquad \text{RWP-STORE}$$

$$\forall n \leq N.\, \Phi(n) \vdash \mathsf{rwp}_{\mathcal{E}}\, \mathsf{rand}\, N\, \{\Phi\} \qquad \text{RWP-RAND}$$

$$\Phi(v) \vdash \mathsf{rwp}_{\mathcal{E}}\, v\, \{\Phi\} \qquad \text{RWP-VAL}$$

$$\mathsf{rwp}_{\mathcal{E}}\, e\, \big\{v.\mathsf{rwp}_{\mathcal{E}}\, K[v]\, \{\Phi\}\big\} \vdash \mathsf{rwp}_{\mathcal{E}}\, K[e]\, \{\Phi\} \qquad \text{RWP-BIND}$$

$$(\forall v.\, \Psi(v) \mathrel{-\!\!*} \Phi(v)) * \mathsf{rwp}_{\mathcal{E}}\, e\, \{\Psi\} \vdash \mathsf{rwp}_{\mathcal{E}}\, e\, \{\Phi\} \qquad \text{RWP-MONO}$$

$$P * \mathsf{rwp}_{\mathcal{E}}\, e\, \{\Phi\} \vdash \mathsf{rwp}_{\mathcal{E}}\, e\, \{v.\, P * \Phi(v)\} \qquad \text{RWP-FRAME}$$

$$\forall \iota.\, \iota \hookrightarrow (N, \epsilon) \mathrel{-\!\!*} \Phi(\iota) \vdash \mathsf{rwp}_{\mathcal{E}}\, \mathsf{tape}\, N\, \{\Phi\} \qquad \text{RWP-TAPE-ALLOC}$$

$$(\iota \hookrightarrow (N, \vec{n}) \mathrel{-\!\!*} \Phi(n)) * \iota \hookrightarrow (N, n \cdot \vec{n}) \vdash \mathsf{rwp}_{\mathcal{E}}\, \mathsf{rand}\, N\, \iota\, \{\Phi\} \qquad \text{RWP-TAPE-EMPTY}$$

$$(\forall n \leq N.\, \iota \hookrightarrow (N, \epsilon) \mathrel{-\!\!*} \Phi(n)) * \iota \hookrightarrow (N, \epsilon) \vdash \mathsf{rwp}_{\mathcal{E}}\, \mathsf{rand}\, N\, \iota\, \{\Phi\} \qquad \text{RWP-TAPE}$$

Lemma A.2 (Model rules).

RWP-COUPL-RAND

$$\frac{\mathsf{step}(m_1) \lesssim \mathsf{unif}(N) : R \qquad \vdash \forall (m_2, n) \in R.\, (\mathsf{spec}(m_2) * P) \mathrel{-\!\!*} \mathsf{rwp}_{\mathcal{E}}\, n\, \{\Phi\}}{\mathsf{spec}(m_1) * \triangleright P \vdash \mathsf{rwp}_{\mathcal{E}}\, \mathsf{rand}\, N\, \{\Phi\}}$$

RWP-SPEC-DET

$$\frac{\mathsf{step}(m_1)(m_2) = 1 \qquad \mathsf{spec}(m_2) * P \vdash \mathsf{rwp}_{\mathcal{E}}\, e\, \{\Phi\}}{\mathsf{spec}(m_1) * \triangleright P \vdash \mathsf{rwp}_{\mathcal{E}}\, e\, \{\Phi\}}$$

B Full Definition of Semantic Model

Definition B.1 (Refinement weakest precondition).

$$\begin{aligned}\mathsf{rwp}_{\mathcal{E}}\, e_1\, \{\Phi\} \triangleq \mathrm{lfp}\, W.(&e_1 \in Val * \Rrightarrow_{\mathcal{E}} \Phi(e_1)) \lor \\ &(e_1 \notin Val * \forall \sigma_1, m_1.\, M(m_1) * S(\sigma_1) \mathrel{-\!\!*} {}_{\mathcal{E}}{\Rrightarrow}_{\emptyset} \\ &\quad \mathsf{cpl}\, m_1 \sim (e_1, \sigma_1)\, \big\{m_2, (e_2, \sigma_2).\, {}_{\emptyset}{\Rrightarrow}_{\mathcal{E}} M(m_2) * S(\sigma_2) * W(e_2, \mathcal{E}, \Phi)\big\})\end{aligned}$$

Definition B.2 (Coupling modality).

$$\frac{\mathsf{reducible}(\rho_1) \qquad \mathsf{ret}(m) \lesssim \mathsf{step}(\rho_1) : R \qquad \forall s \in R.\, \Rrightarrow_{\emptyset} \Psi(s)}{\mathsf{cpl}\, m \sim \rho_1\, \{\Psi\}}$$

$$\frac{\mathsf{reducible}(\rho_1) \qquad \mathsf{reducible}(m_1) \qquad \mathsf{step}(m_1) \lesssim \mathsf{step}(\rho_1) : R \qquad \forall s \in R.\, \triangleright \Rrightarrow_{\emptyset} \Psi(s)}{\mathsf{cpl}\, m_1 \sim \rho_1\, \{\Psi\}}$$

$$\frac{\mathsf{reducible}(m_1) \qquad \mathsf{step}(m_1) \lesssim \mathsf{ret}(\rho) : R \qquad \forall (m_2, \rho) \in R.\, \triangleright \Rrightarrow_{\emptyset} \mathsf{cpl}\, m_2 \sim \rho\, \{\Psi\}}{\mathsf{cpl}\, m_1 \sim \rho\, \{\Psi\}}$$

$$\frac{\begin{array}{c}\mathsf{reducible}(m_1) \qquad l \subseteq \mathrm{dom}(\sigma_1) \\ \mathsf{step}(m_1) \lesssim \mathsf{foldM}(\mathsf{sstep}, \sigma_1, l) : R \qquad \forall (m_2, \sigma_2) \in R.\, \triangleright \Rrightarrow_{\emptyset} \mathsf{cpl}\, m_2 \sim (e_1, \sigma_2)\, \{\Psi\}\end{array}}{\mathsf{cpl}\, (e_1, \sigma_1) \sim m_1\, \{\Psi\}}$$

References

Sheshansh Agrawal, Krishnendu Chatterjee, and Petr Novotný. 2018. Lexicographic ranking supermartingales: an efficient approach to termination of probabilistic programs. *Proc. ACM Program. Lang.* 2, POPL (2018), 34:1–34:32. https://doi.org/10.1145/3158122

Alejandro Aguirre, Gilles Barthe, Lars Birkedal, Ales Bizjak, Marco Gaboardi, and Deepak Garg. 2018. Relational Reasoning for Markov Chains in a Probabilistic Guarded Lambda Calculus. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10801)*, Amal Ahmed (Ed.). Springer, 214–241. https://doi.org/10.1007/978-3-319-89884-1_8

Alejandro Aguirre and Lars Birkedal. 2023. Step-Indexed Logical Relations for Countable Nondeterminism and Probabilistic Choice. *Proc. ACM Program. Lang.* 7, POPL (2023), 33–60. https://doi.org/10.1145/3571195

Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. 2007. A very modal model of a modern, major, general type system. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007.* 109–122. https://doi.org/10.1145/1190216.1190235

Tamarah Arons, Amir Pnueli, and Lenore D. Zuck. 2003. Parameterized Verification by Probabilistic Abstraction. In *Foundations of Software Science and Computational Structures, 6th International Conference, FOSSACS 2003 Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2620)*, Andrew D. Gordon (Ed.). Springer, 87–102. https://doi.org/10.1007/3-540-36576-1_6

Krishna B. Athreya and Peter E. Ney. 1972. *Branching Processes.* Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-65371-1

Robert Atkey. 2011. Amortised Resource Analysis with Separation Logic. *Log. Methods Comput. Sci.* 7, 2 (2011). https://doi.org/10.2168/LMCS-7(2:17)2011

Martin Avanzini, Gilles Barthe, and Ugo Dal Lago. 2021. On continuation-passing transformations and expected cost analysis. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–30. https://doi.org/10.1145/3473592

Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. 2008. Formal certification of code-based cryptographic proofs. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '09.* ACM Press, Savannah, GA, USA, 90. https://doi.org/10.1145/1480881.1480894

Lars Birkedal and Aleš Bizjak. 2023. Lecture Notes on Iris: Higher-Order Concurrent Separation Logic. http://iris-project.org/tutorial-pdfs/iris-lecture-notes.pdf

Lars Birkedal, Rasmus Ejlers Møgelberg, Jan Schwinghammer, and Kristian Støvring. 2012. First steps in synthetic guarded domain theory: step-indexing in the topos of trees. *Log. Methods Comput. Sci.* 8, 4 (2012). https://doi.org/10.2168/LMCS-8(4:1)2012

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

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

Olivier Bournez and Florent Garnier. 2005. Proving Positive Almost-Sure Termination. In *Term Rewriting and Applications, 16th International Conference, RTA 2005, Nara, Japan, April 19-21, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3467)*, Jürgen Giesl (Ed.). Springer, 323–337. https://doi.org/10.1007/978-3-540-32033-3_24

Tomás Brázdil, Javier Esparza, Stefan Kiefer, and Antonín Kucera. 2013. Analyzing probabilistic pushdown automata. *Formal Methods Syst. Des.* 43, 2 (2013), 124–163. https://doi.org/10.1007/S10703-012-0166-0

Aleksandar Chakarov and Sriram Sankaranarayanan. 2013. Probabilistic Program Analysis with Martingales. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings.* 511–526. https://doi.org/10.1007/978-3-642-39799-8_34

Arthur Charguéraud and François Pottier. 2019. Verifying the Correctness and Amortized Complexity of a Union-Find Implementation in Separation Logic with Time Credits. *J. Autom. Reason.* 62, 3 (2019), 331–365. https://doi.org/10.1007/S10817-017-9431-7

Donald L. Cohn. 2013. *Measure Theory: Second Edition.* Springer New York. https://doi.org/10.1007/978-1-4614-6956-8

Coq Development Team. 2023. *The Coq Proof Assistant.* https://doi.org/10.5281/zenodo.8161141

Luc Devroye. 2012. Simulating Size-constrained Galton-Watson Trees. *SIAM J. Comput.* 41, 1 (2012), 1–11. https://doi.org/10.1137/090766632

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

Manuel Eberl, Max W. Haslbeck, and Tobias Nipkow. 2020. Verified Analysis of Random Binary Tree Structures. *J. Autom. Reason.* 64, 5 (2020), 879–910. https://doi.org/10.1007/S10817-020-09545-0

Jeff Erickson. 2017. Treaps and Skip Lists. https://jeffe.cs.illinois.edu/teaching/algorithms/notes/03-treaps.pdf Lecture notes.

Javier Esparza, Andreas Gaiser, and Stefan Kiefer. 2012. Proving Termination of Probabilistic Programs Using Patterns. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings (Lecture Notes in Computer Science, Vol. 7358)*, P. Madhusudan and Sanjit A. Seshia (Eds.). Springer, 123–138. https://doi.org/10.1007/978-3-642-31424-7_14

Kousha Etessami and Mihalis Yannakakis. 2009. Recursive Markov chains, stochastic grammars, and monotone systems of nonlinear equations. *J. ACM* 56, 1 (2009), 1:1–1:66. https://doi.org/10.1145/1462153.1462154

Shenghua Feng, Mingshuai Chen, Han Su, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Naijun Zhan. 2023. Lower Bounds for Possibly Divergent Probabilistic Programs. *Proc. ACM Program. Lang.* 7, OOPSLA1 (2023), 696–726. https://doi.org/10.1145/3586051

Luis María Ferrer Fioriti and Holger Hermanns. 2015. Probabilistic Termination: Soundness, Completeness, and Compositionality. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 489–501. https://doi.org/10.1145/2676726.2677001

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

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

Hongfei Fu and Krishnendu Chatterjee. 2019. Termination of Nondeterministic Probabilistic Programs. In *Verification, Model Checking, and Abstract Interpretation - 20th International Conference, VMCAI 2019, Cascais, Portugal, January 13-15, 2019, Proceedings.* 468–490. https://doi.org/10.1007/978-3-030-11245-5_22

Simon Oddershede Gregersen, Alejandro Aguirre, Philipp G. Haselwarter, Joseph Tassarotti, and Lars Birkedal. 2024a. *Almost-Sure Termination by Guarded Refinement - Coq Artifact.* https://doi.org/10.5281/zenodo.11481248

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

Simon Oddershede Gregersen, Johan Bay, Amin Timany, and Lars Birkedal. 2021. Mechanized Logical Relations for Termination-Insensitive Noninterference. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–29. https://doi.org/10.1145/3434291

Sergiu Hart, Micha Sharir, and Amir Pnueli. 1983. Termination of Probabilistic Concurrent Program. *ACM Trans. Program. Lang. Syst.* 5, 3 (1983), 356–380. https://doi.org/10.1145/2166.357214

Joe Hurd. 2002. A Formal Approach to Probabilistic Termination. In *Theorem Proving in Higher Order Logics, 15th International Conference, TPHOLs 2002, Hampton, VA, USA, August 20-23, 2002, Proceedings (Lecture Notes in Computer Science, Vol. 2410)*, Victor Carreño, César A. Muñoz, and Sofiène Tahar (Eds.). Springer, 230–245. https://doi.org/10.1007/3-540-45685-6_16

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

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

Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. 2019. On the hardness of analyzing probabilistic programs. *Acta Informatica* 56, 3 (2019), 255–285. https://doi.org/10.1007/S00236-018-0321-1

Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. 2016. Weakest Precondition Reasoning for Expected Run-Times of Probabilistic Programs. In *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings.* 364–389. https://doi.org/10.1007/978-3-662-49498-1_15

Naoki Kobayashi, Ugo Dal Lago, and Charles Grellois. 2019. On the Termination Problem for Probabilistic Higher-Order Recursive Programs. In *34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019, Vancouver, BC, Canada, June 24-27, 2019.* 1–14. https://doi.org/10.1109/LICS.2019.8785679

Naoki Kobayashi, Ugo Dal Lago, and Charles Grellois. 2020. On the Termination Problem for Probabilistic Higher-Order Recursive Programs. *Log. Methods Comput. Sci.* 16, 4 (2020). https://lmcs.episciences.org/6817

Marta Z. Kwiatkowska, Gethin Norman, and David Parker. 2011. PRISM 4.0: Verification of Probabilistic Real-Time Systems. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings.* 585–591. https://doi.org/10.1007/978-3-642-22110-1_47

Ugo Dal Lago, Claudia Faggian, and Simona Ronchi Della Rocca. 2021. Intersection types and (positive) almost-sure termination. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–32. https://doi.org/10.1145/3434313

Ugo Dal Lago and Charles Grellois. 2017. Probabilistic Termination by Monadic Affine Sized Typing. 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 (Lecture Notes in Computer Science, Vol. 10201)*, Hongseok Yang (Ed.). Springer, 393–419. https://doi.org/10.1007/978-3-662-54434-1_15

P. J. Landin. 1964. The Mechanical Evaluation of Expressions. *Comput. J.* 6, 4 (1964), 308–320. https://doi.org/10.1093/COMJNL/6.4.308

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

Rupak Majumdar and V. R. Sathiyanarayana. 2024. Positive Almost-Sure Termination: Complexity and Proof Rules. *Proc. ACM Program. Lang.* 8, POPL (2024), 1089–1117. https://doi.org/10.1145/3632879

Annabelle McIver, Carroll Morgan, Benjamin Lucien Kaminski, and Joost-Pieter Katoen. 2018. A new proof rule for almost-sure termination. *Proc. ACM Program. Lang.* 2, POPL (2018), 33:1–33:28. https://doi.org/10.1145/3158121

Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2019. Time Credits and Time Receipts in Iris. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings.* 3–29. https://doi.org/10.1007/978-3-030-17184-1_1

Carroll Morgan and Annabelle McIver. 1999. pGCL: formal reasoning for random algorithms. *South African Computer Journal* 22 (1999), 14–27.

Hiroshi Nakano. 2000. A Modality for Recursion. In *15th Annual IEEE Symposium on Logic in Computer Science, Santa Barbara, California, USA, June 26-29, 2000.* 255–266. https://doi.org/10.1109/LICS.2000.855774

Van Chan Ngo, Quentin Carbonneaux, and Jan Hoffmann. 2018. Bounded expectations: resource analysis for probabilistic programs. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 496–512. https://doi.org/10.1145/3192366.3192394

François Pottier, Armaël Guéneau, Jacques-Henri Jourdan, and Glen Mével. 2024. Thunks and Debits in Separation Logic with Time Credits. *Proc. ACM Program. Lang.* 8, POPL (2024), 1482–1508. https://doi.org/10.1145/3632892

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings.* IEEE Computer Society, 55–74. https://doi.org/10.1109/LICS.2002.1029817

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

Simon Spies, Lennard Gäher, Daniel Gratzer, Joseph Tassarotti, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2021a. Transfinite Iris: resolving an existential dilemma of step-indexed separation logic. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021.* 80–95. https://doi.org/10.1145/3453483.3454031

Simon Spies, Neel Krishnaswami, and Derek Dreyer. 2021b. Transfinite step-indexing for termination. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–29. https://doi.org/10.1145/3434294

Frank Spitzer. 1964. *Principles of Random Walk.* Springer New York. https://doi.org/10.1007/978-1-4757-4229-9

Kasper Svendsen, Filip Sieczkowski, and Lars Birkedal. 2016. Transfinite Step-Indexing: Decoupling Concrete and Logical Steps. In *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9632)*, Peter Thiemann (Ed.). Springer, 727–751. https://doi.org/10.1007/978-3-662-49498-1_28

Alfred Tarski. 1955. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.* 5, 2 (June 1955), 285–309. https://doi.org/10.2140/pjm.1955.5.285

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

Joseph Tassarotti, Ralf Jung, and Robert Harper. 2017. A Higher-Order Logic for Concurrent Termination-Preserving Refinement. 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 (Lecture Notes in Computer Science, Vol. 10201)*, Hongseok Yang (Ed.). Springer, 909–936. https://doi.org/10.1007/978-3-662-54434-1_34

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

Amin Timany, Simon Oddershede Gregersen, Léo Stefanesco, Jonas Kastberg Hinrichsen, Léon Gondelman, Abel Nieto, and Lars Birkedal. 2024a. Trillium: Higher-Order Concurrent and Distributed Separation Logic for Intensional Refinement. *Proc. ACM Program. Lang.* 8, POPL (2024), 241–272. https://doi.org/10.1145/3632851

Amin Timany, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2024b. A Logical Approach to Type Soundness. (2024). https://iris-project.org/pdfs/2024-jacm-logical-type-soundness.pdf Manuscript.

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

Cédric Villani. 2009. *Optimal Transport.* Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-71050-9

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

Di Wang, David M. Kahn, and Jan Hoffmann. 2020. Raising expectations: automating expected cost analysis with types. *Proc. ACM Program. Lang.* 4, ICFP (2020), 110:1–110:31. https://doi.org/10.1145/3408992