Error Credits: Resourceful Reasoning about Error Bounds for Higher-Order Probabilistic Programs

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

Probabilistic programs often trade accuracy for efficiency, and thus may, with a small probability, return an incorrect result. It is important to obtain precise bounds for the probability of these errors, but existing verification approaches have limitations that lead to error probability bounds that are excessively coarse, or only apply to first-order programs. In this paper we present Eris, a higher-order separation logic for proving error probability bounds for probabilistic programs written in an expressive higher-order language.

Our key novelty is the introduction of *error credits*, a separation logic resource that tracks an upper bound on the probability that a program returns an erroneous result. By representing error bounds as a resource, we recover the benefits of separation logic, including compositionality, modularity, and dependency between errors and program terms, allowing for more precise specifications. Moreover, we enable novel reasoning principles such as expectation-preserving error composition, amortized error reasoning, and error induction.

We illustrate the advantages of our approach by proving amortized error bounds on a range of examples, including collision probabilities in hash functions, which allow us to write more modular specifications for data structures that use them as clients. We also use our logic to prove correctness and almost-sure termination of rejection sampling algorithms. All of our results have been mechanized in the Coq proof assistant using the Iris separation logic framework and the Coquelicot real analysis library.

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

Additional Key Words and Phrases: error bounds, error credits, almost-sure termination

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

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

Authors' Contact Information: Alejandro Aguirre, Aarhus University, Denmark, alejandro@cs.au.dk; Philipp G. Haselwarter, Aarhus University, Denmark, pgh@cs.au.dk; Markus de Medeiros, New York University, USA, mjd9606@nyu.edu; Kwing Hei Li, Aarhus University, Denmark, hei.li@cs.au.dk; Simon Oddershede Gregersen, New York University, USA, s.gregersen@nyu.edu; Joseph Tassarotti, New York University, USA, jt4767@nyu.edu; Lars Birkedal, Aarhus University, Denmark, birkedal@cs.au.dk.

ACM 2475-1421/2024/8-ART246
https://doi.org/10.1145/3674635

1 Introduction

Randomness is an important tool in the design of efficient algorithms and data structures and is widely used in many application domains, including cryptography and machine learning. In many cases, probabilistic programs are only approximately correct because they admit unwanted behaviors with low probability, usually as a trade-off for better performance.

One major class of randomized algorithms are so-called Monte Carlo algorithms; they admit wrong results with small probability, but are usually much faster than their deterministic counterparts. For example, primality tests such as Miller-Rabin [Miller 1975; Rabin 1980] or Solovay-Strassen [Solovay and Strassen 1977] are Monte Carlo algorithms: they check the divisibility of a potential prime by a number of randomly selected candidates and answer in polynomial time with either "probably prime" (which may happen for composite inputs with low probability) or with "certainly composite". The other major class are Las Vegas algorithms. These never return wrong results, but their running time is a random variable which may take high values but generally with low probability. For example, a rejection sampler searching for a "good" sample in a large sample space might give up after a bounded number of iterations if no good sample can be found in time (Monte Carlo) or continue searching indefinitely (Las Vegas).

The trade-off between efficiency and accuracy/termination is typically justified by observing that the unwanted behavior occurs only with small probability. Establishing bounds on the probability of these errors (from now on, *"error bounds"*) is therefore an important prerequisite for the use of Monte Carlo and Las Vegas algorithms. However, probabilistic reasoning is often counterintuitive and when combined with reasoning about the complex state space of probabilistic programs, it quickly becomes infeasible to manually establish error bounds for even moderately complicated programs. To address this problem, probabilistic program logics offer a rigorous way to establish trust in the correctness of randomized programs. For randomized first-order `WHILE` programs, the approximate Hoare logic (aHL) of Barthe et al. [2016b] provides a convenient way to over-approximate the error behavior of an algorithm. Formally, aHL judgments are annotated with an "error budget" ε. The judgment $\vdash_{\varepsilon} \{P\}\, c\, \{Q\}$ means that when the precondition P holds, the probability that the postcondition Q is *violated* after executing c is at most ε. The logic supports local reasoning through *union bounds*:[1] the error of a sequence of commands $c_1; c_2$ is bounded by $\varepsilon_1 + \varepsilon_2$ if c_1 (resp. c_2) has error ε_1 (resp. ε_2) when considered in isolation. This principle is formalized in the rule for sequential composition:

$$\frac{\vdash_{\varepsilon_1} \{P\}\, c_1\, \{Q\} \qquad \vdash_{\varepsilon_2} \{Q\}\, c_2\, \{R\}}{\vdash_{\varepsilon_1+\varepsilon_2} \{P\}\, c_1; c_2\, \{R\}}\ \textsc{aHL seq}$$

Subsequent work [Aguirre et al. 2021; Sato et al. 2019] develops a higher-order union bound logic (HO-UBL) for a monadic presentation of a probabilistic λ-calculus without recursion.

However, by baking the error bounds into the judgmental structure of the logic rather than treating them as ordinary propositions, these works on approximate correctness forego some of the ability to reason about errors in a modular way. For instance, an error bound in aHL [Barthe et al. 2016b] cannot depend on a program term, and HO-UBL [Aguirre et al. 2021, §4.1] cannot prove the expected approximate higher-order specifications for simple functions such as List.iter because the error is a part of the judgement, and not a first class proposition which may itself occur in, *e.g.*, a pre- or postcondition.

Furthermore, reasoning about composition via union bounds over-approximates error and can produce excessively coarse bounds when errors are not independent. Specifically, the union

[1] The principle behind union bounds is also known as "Boole's inequality".

bound for two events A and B bounds the probability $\Pr[A \vee B] = \Pr[A] + \Pr[B] - \Pr[A \wedge B]$ by $\Pr[A] + \Pr[B]$, thereby losing precision when $\Pr[A \wedge B]$ is large.

In this paper, we present Eris: a higher-order separation logic with support for advanced reasoning principles for proving bounds on the probabilities of errors for programs written in $\lambda^{\mathrm{rand}}_{\mathrm{ref}}$, an expressive ML-like language with random sampling, full recursion, higher-order functions, and higher-order store. Inspired by time credits [Atkey 2011; Charguéraud and Pottier 2019; Mével et al. 2019] to reason about cost as a resource, we introduce and develop *error credits*. Ownership of ε error credits is a first-class proposition in Eris, written $\lightning(\varepsilon)$ (read: "up to ε"), and proving an Eris specification $\vdash \{P * \lightning(\varepsilon)\}\ e\ \{Q\}$ intuitively means: "if P holds then the probability of e crashing or returning a result that violates Q is at most ε".

The resource approach affords Eris great flexibility: if we own an error credit $\lightning(\varepsilon)$, we can choose to spend it however suits our proof. We can "pay" for an operation which fails with probability ε, store it in an invariant describing a probabilistic data structure, frame it away during a function call to keep it for later, or split it into any number of credits $\lightning(\varepsilon_1), \ldots, \lightning(\varepsilon_n)$ so long as we stay below the initial error budget, *i.e.* $\sum_{i=1}^{n} \varepsilon_i \leq \varepsilon$.

We now proceed to outline how Eris addresses the two limitations of prior work mentioned above (lack of modularity and conservative error bounds), and afterwards we explain how the error credits of Eris also support a novel form of reasoning about *amortized error bounds* for randomized data structures. Finally, we give an overview of how error credits in a *total correctness* version of Eris can be used to prove *almost-sure termination* of Las Vegas algorithms.

Modular specifications of higher-order programs. Eris can be used to give modular specifications to higher-order functions. For a concrete example, consider the specification below for a (parallel) iterator function List.iter, which is just the standard specification one would prove in a non-probabilistic setting. In the precondition, the first line states that the argument l is program-level representation for the mathematical sequence of values xs. Then, the second line assumes a specification for the function e to be iterated over the list: for each argument x, e takes $P(x)$ as a precondition and returns $Q(x)$ as a postcondition. The last line states that the precondition P holds for each x in the list xs. Finally, the postcondition states that Q holds for each x in xs.

$$\left\{ \begin{array}{c} \mathit{isList}\ xs\ l\ * \\ \forall x.\ \{P(x)\}\ e\ x\ \{Q(x)\}\ * \\ \bigstar_{x \in xs} P(x) \end{array} \right\} \mathsf{List.iter}\ e\ l \left\{ \mathop{\bigstar}_{x \in xs} Q(x) \right\}$$

By instantiating the higher-order specification, we can reason in a setting in which the specification of e only holds up to some error bound as shown below. Now, the second line states that assuming that the precondition $R(x)$ holds, the postcondition $Q(x)$ will hold except with some error probability $\mathcal{E}(x)$. Notably, the error can depend on the value of the argument x. The third line requires that the user of the List.iter has have enough error credits to "pay" for each call to e.

$$\left\{ \begin{array}{c} \mathit{isList}\ xs\ l\ * \\ \forall x.\ \left\{R(x) * \lightning(\mathcal{E}(x))\right\}\ e\ x\ \{Q(x)\}\ * \\ \bigstar_{x \in xs} \left(R(x) * \lightning(\mathcal{E}(x))\right) \end{array} \right\} \mathsf{List.iter}\ e\ l \left\{ \mathop{\bigstar}_{x \in xs} Q(x) \right\}$$

It is noteworthy that by treating error as a resource, the version *with* error credits can be *derived* from the standard version by instantiating $P(x)$ to be $R(x) * \lightning(\mathcal{E}(x))$. The specification of List.iter does not need to be reproven and the probabilistic version just becomes a special case.

More precise error bounds via expectation-preserving composition. In addition to bringing union-bound reasoning in the style of Aguirre et al. [2021]; Barthe et al. [2016b]; Sato et al. [2019] to a richer programming language, Eris supports a novel form of reasoning about errors *in expectation*, which leads to more precise error bounds. This feature hinges on two observations.

The first observation is a simple but useful consequence of treating errors bounds as separation logic resources: as mentioned above, error bounds can depend on values of computations. Concretely, consider the following instantiation of the so-called *bind* rule:

$$\frac{\vdash \{P * ↯(\varepsilon_1)\}\ e_1\ \{x.\, ↯(\mathcal{E}_2(x))\ *\ Q\} \qquad \vdash \forall x.\ \{Q * ↯(\mathcal{E}_2(x))\}\ e_2\ \{R\}}{\vdash \{P * ↯(\varepsilon_1)\}\ \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\ \{R\}}\ \textsc{ht-bind-exp}$$

The rule expresses that starting with ε_1 credits to begin with, if all evaluations of e_1 leave enough credits $\mathcal{E}_2(x)$ to verify the continuation R, then ε_1 credits suffice to verify the entire let expression. Recall that the x. in the postcondition of e_1 acts as a binder that captures the return value of the computation. By this rule, Eris supports *value-dependent error composition*: in different branches of e_2 we can spend different amounts of credits depending on x, giving us a more precise error bound than the maximum error across all cases. We present a concrete example of this phenomenon in §3.

The second observation is that, whenever the program takes a step and we initially have ε_1 credits, then we can split our ε_1 error credits across all possible branches as a *weighted sum* according to the probability of each branch. For example, if we sample i uniformly from the set $\{0, \ldots, N\}$, and moreover we know that any continuation needs $\mathcal{E}_2(i)$ credits, then it suffices to have $\mathbb{E}[\mathcal{E}_2] = \sum_{i=0}^{N} \mathcal{E}_2(i)/(N+1)$ credits at the start. This is captured formally in the proof rule for the random sampling operation shown below:

$$\frac{\sum_{i=0}^{N} \dfrac{\mathcal{E}_2(i)}{N+1} = \varepsilon_1}{\vdash \{↯(\varepsilon_1)\}\ \mathsf{rand}\ N\ \{n.\, ↯(\mathcal{E}_2(n))\}}\ \textsc{ht-rand-exp}$$

To put the two observations to use, let us consider a concrete example of a composite computation for which we get more precise error bounds than would be possible in previous work:

$$\left\{\begin{array}{c} \forall x.\ \{\varepsilon\}\ e\ 0\ \{\mathsf{True}\} \ * \\ ↯\left(\varepsilon \cdot \frac{K}{2}\right) \end{array}\right\} \begin{array}{l} \mathsf{let}\ n = \mathsf{rand}\ K\ \mathsf{in} \\ \mathsf{let}\ l = \mathsf{List.make}\ 0\ n\ \mathsf{in}\ \mathsf{List.iter}\ e\ l \end{array} \ \{\mathsf{True}\}$$

Here we intend List.make $0\ n$ to construct a list of zeros of length n, where n is sampled uniformly from the set $\{0, \ldots, K\}$, and that e is a function to be iterated, such that it with the argument 0 requires ε credits to execute safely (i.e. without crashing). Using the ht-bind-exp rule, we can then prove that the composed program executes safely if we have $\varepsilon \cdot \frac{K}{2}$ credits in our precondition and set $\mathcal{E}_2(n) = n \cdot \varepsilon$ in the postcondition of rand K. By the specification of List.iter we will need $↯(\varepsilon)$ for every element of the list we iterate over, that is $↯(\varepsilon \cdot n)$ for a list of length n. Note that $\varepsilon \cdot \frac{K}{2}$ is precisely the expected value of $\mathcal{E}_2$, and in combination, the two observations mean that we obtain a form of *expectation-preserving composition*. We further discuss these rules in §3.

Amortized error bounds. By representing error bounds as a resource, Eris not only addresses the limitations of prior work mentioned above, but also supports reasoning about *amortized error bounds* for operations of randomized data structures. Inspired by the work on type-based resource analysis of Hofmann and Jost [2003], Atkey [2011] pioneered the use of "time credits" to reason about amortized time complexity in separation logic, and the idea was subsequently extended and formalized in different separation logics [Charguéraud and Pottier 2019; Mével et al. 2019]. Our use of error credits in turn allows us to give modular, amortized specifications of randomized data structures which hide implementation details such as the timing of "costly" (*i.e.* error-prone) internal operations. In §4 we present several case studies that demonstrate how Eris supports modular reasoning about amortized error bounds.

Almost-sure termination using error credits. So far, we have implicitly considered a partial correctness interpretation of the Eris Hoare triples. In particular, this means that a divergent program trivially satisfies any Hoare triple; in Eris this can be proven using so-called guarded recursion/Löb induction. This proof principle is sound in Eris because the semantics of Hoare triples is defined using a guarded fixed point. Now, an interesting observation is that we can easily make a "total-correctness" version of Eris called Eris_t by instead defining the semantics of Hoare triples as a *least fixed point*. Because of the approximate up-to-error interpretation of Hoare triples this yields an "approximate total-correctness" interpretation: a Hoare triple with ε credits in the precondition bounds the probability of never reaching a value satisfying the postcondition, which includes both the possibility of not satisfying the postcondition and the possibility of diverging. In particular, the probability of termination is *at least* $1 - \varepsilon$. In turn, if we can show a total Hoare triple for *an arbitrary* error bound ε, then we can conclude that it holds when ε becomes vanishingly small and thus that the program almost-surely terminates, i.e. it terminates with probability 1. We state and prove these properties more formally in §5 and §6; the soundness of this approach relies on a continuity argument for the semantics. To the best of our knowledge, this argument and the approach of showing almost-sure termination via error credits is novel. We demonstrate in §5.2 how it can be used to prove correctness of several Las Vegas algorithms, including rejection samplers.

Contributions. To summarize, we provide:

- The first probabilistic higher-order separation logic, Eris, for *modular* approximate reasoning (up-to-errors) about probabilistic programs written in $\lambda^{\text{rand}}_{\text{ref}}$, a randomized higher-order language with higher-order references.
- A resourceful account of errors which allows for more precise accounting of error bounds via value-dependent and expectation-preserving composition, and for reasoning about a richer class of properties, in particular amortized error bounds.
- A total correctness version of Eris, which can be used to establish *lower* bounds on probabilities of program behaviors and thus to prove almost-sure termination.
- A substantial collection of case studies which demonstrate how the proof principles mentioned work in practice.
- All of the results in this paper have been mechanized (see [Aguirre et al. 2024]) in the Coq proof assistant, building on the Iris separation logic framework [Jung et al. 2016, 2018, 2015a; Krebbers et al. 2017] and the Coquelicot real analysis library [Boldo et al. 2015].

Outline. In §2 we recall some preliminaries and define the operational semantics of $\lambda^{\text{rand}}_{\text{ref}}$. We then introduce Eris in §3. We demonstrate how to use Eris on a range of case studies, focusing on amortized error bounds, in §4. Afterwards, in §5 we describe how a total version Eris_t of Eris can be used to reason about almost-sure termination via error credits; the section includes a number of case studies (and more can be found in the long version of the paper[2]). Then we present the model of Eris in §6 and sketch how the model is used to prove soundness and adequacy of the logics. Finally, we discuss related work in §7 and conclude and discuss future work in §8.

2 Preliminaries and the Language $\lambda^{\text{rand}}_{\text{ref}}$

In §2.1 we first recall elements of discrete probability theory required to define the semantics of our probabilistic language $\lambda^{\text{rand}}_{\text{ref}}$ and introduce the definitions we use to express approximate reasoning. We subsequently define the syntax and operational semantics of $\lambda^{\text{rand}}_{\text{ref}}$ in §2.2.

[2]A full version of the paper with appendix can be found at https://arxiv.org/pdf/2404.14223

2.1 Probabilities and Programs

As a first approximation, one might expect the execution of a randomized program e to produce a (discrete) probability distribution on values. However, since programs may not terminate, programs might not induce *proper* distributions, but rather *subdistributions*, whose total mass is upper-bounded by 1, but may be lower.

Definition 1 (Mass). *Let $\mathbb{R}^{\geq 0}$ denote the non-negative real numbers. For a countable set X, the* mass *of a function $f : X \rightarrow \mathbb{R}^{\geq 0}$ is given by $|f| = \sum_{x \in X} f(x)$ if this sum is finite.*

Definition 2 (Subdistribution). *A* (discrete) probability subdistribution *on a countable set X is a function $\mu : X \rightarrow [0, 1]$ such that $|\mu| \leq 1$. We say that μ is a* proper *probability distribution if $|\mu| = 1$. We write $\mathcal{D}(X)$ for the set of all subdistributions on X.*

We simply write *distribution* to mean "discrete probability subdistribution" in the remainder of the paper. Unless otherwise specified, the variable μ denotes a distribution, and X or Y a countable set, typically the set of values, expressions, or configurations of $\lambda^{\text{rand}}_{\text{ref}}$.

Lemma 3 (Probability Monad). *Let $\mu \in \mathcal{D}(X)$, $x \in X$, and $f : X \rightarrow \mathcal{D}(Y)$. Then*

(1) $\mathrm{bind}(f, \mu)(y) \triangleq \sum_{x \in X} \mu(x) \cdot f(x)(y)$

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

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

Definition 4 (Restriction). *Let P be a predicate on X. The* restriction of μ to P *is given by:*

$$\mu_{|P}(x) = \begin{cases} \mu(x) & \text{if } P(x) \text{ holds,} \\ 0 & \text{otherwise.} \end{cases}$$

Definition 5 (Probability of a Predicate). *The* probability of a predicate P with respect to μ, *written as $\Pr_\mu[P]$, is the total probability mass of μ satisfying P,* i.e. $\Pr_\mu[P] = |\mu_{|P}|$.

2.2 Language Definition and Operational Semantics

The syntax of $\lambda^{\text{rand}}_{\text{ref}}$, the language we consider in this paper, is defined by the grammar below.

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

The term language is mostly standard: $\mathsf{allocn}\ e_1\ e_2$ allocates a new array of length e_1 with each cell containing the value returned by e_2, $!\, e$ 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 $(\lambda x.\, e_2)\ e_1$, sequencing $e_1; e_2$ defined as $\mathsf{let}\ \_ = e_1\ \mathsf{in}\ e_2$, and references $\mathsf{ref}\ e$ defined as $\mathsf{allocn}\ e\ 1$. We write $l[b]$ as sugar for offsetting location l by b, defined as $(l + b)$.

Our language matches that of Clutch [Gregersen et al. 2024], modulo the minor difference that we add arrays.[3] States in $\lambda^{\text{rand}}_{\text{ref}}$ are finite maps from memory locations to values.

[3]As in Clutch, we support presampling tapes, but since they are not used until §5, we relegate this discussion to §5.2.3.

To define full program execution, we define $\mathrm{step}(\rho) \in \mathcal{D}(\mathit{Cfg})$, the distribution induced by the single step reduction of configuration $\rho \in \mathit{Cfg}$. The semantics of step is standard: all non-probabilistic constructs reduce deterministically as usual, *e.g.*, $\mathrm{step}(\mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2, \sigma) = \mathrm{ret}(e_1, \sigma)$, and the probabilistic choice operator $\mathsf{rand}\ N$ reduces uniformly at random:

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

The Boolean operation flip is syntactic sugar for $(\mathsf{rand}\ 1 == 1)$.

With the single step reduction step defined, we now define a stratified execution probability $\mathrm{exec}_n : \mathit{Cfg} \longrightarrow \mathcal{D}(\mathit{Val})$ by induction on n:

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

where $\mathbf{0}$ denotes the everywhere-zero distribution. The probability that a full execution, starting from configuration ρ, reaches a value v is taken as the limit of its stratified approximations, which exists by monotonicity and boundedness:

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

We simply write $\mathrm{exec}\ e$ as notation for $\mathrm{exec}(e, \sigma)$ if $\mathrm{exec}(e, \sigma)$ is the same for all states σ.

As an example, consider the program $e \triangleq \mathsf{if}\ \mathsf{flip}\ \&\&\ \mathsf{flip}\ \mathsf{then}\ 42\ \mathsf{else}\ \Omega$, where Ω is a diverging term and && denotes logical conjunction. If we execute e, we either obtain the value 42 in a few steps (both flips return true with probability $0.5 \times 0.5 = 0.25$), or we do not obtain a value at all otherwise. In other words, $\mathrm{exec}\ e$ induces the subdistribution $\{42 \mapsto 0.25,\ \_ \mapsto 0\} : \mathit{Val} \longrightarrow [0, 1]$.

3 The Eris Logic

In this section we introduce the Eris logic. We first present the propositions of Eris, and then to provide some intuition for the program logic proof rules we present the adequacy theorem, which expresses what one can conclude by proving a Hoare triple in Eris. The adequacy theorem itself is only proved later (§6.2) when we introduce the semantic model of Eris. After the adequacy theorem, we then present a selection of the program logic rules of Eris.

Eris is based on the Iris separation logic framework [Jung et al. 2018] and inherits all of the basic propositions and their associated proof rules. An excerpt of Eris propositions is shown below, including the later modality $\triangleright$, the persistence modality $\square$ and the points-to connective $\ell \mapsto v$, which asserts ownership of the location ℓ and its content v:

$$\begin{aligned} P, Q \in \mathit{iProp} ::= \mathsf{True} \mid \mathsf{False} \mid P \land Q \mid P \lor Q \mid P \Rightarrow Q \mid \forall x.\, P \mid \exists x.\, P \mid P * Q \mid P \mathrel{-\!\!*} Q \mid \\ \square P \mid \triangleright P \mid \ell \mapsto v \mid ⨍(\varepsilon) \mid \{P\}\ e\ \{Q\} \mid \dots \end{aligned}$$

The main novelty of Eris is the program logic $\{P\}\ e\ \{Q\}$ and the new $⨍(\varepsilon)$ assertion which denotes ownership of ε error credits. Error credits satisfy the following rules:

$$⨍(\varepsilon_1) * ⨍(\varepsilon_2) \dashv\vdash ⨍(\varepsilon_1 + \varepsilon_2) \qquad ⨍(\varepsilon_1) * (\varepsilon_2 < \varepsilon_1) \vdash ⨍(\varepsilon_2) \qquad ⨍(1) \vdash \mathsf{False}$$

From the point of view of a user of the logic, this interface (and the rule ht-rand-exp below) is all they need to know about credits. The first rule expresses that ownership of $\varepsilon_1 + \varepsilon_2$ error credits is the same as ownership of ε_1 credits and ownership of ε_2 credits. The second rule says that it is sound to throw away credits that we own. Finally, the last rule says that if we own 1 full error credit, then we can immediately conclude a contradiction. Intuitively, ownership of $⨍(1)$ corresponds to proving a statement holds with probability at least 0, which is trivially true.

We define the semantics of the program logic in §6. The adequacy theorem shown below captures what a specification with error credits means in terms of probabilities and the operational semantics.

Theorem 6 (Adequacy). *If* $\vdash \{↯(\varepsilon)\}\ e\ \{\phi\}$ *then* $\mathrm{Pr}_{\mathrm{exec}\, e}[\neg\phi] \leq \varepsilon$. *Moreover, the probability of e getting stuck is at most* ε.

The adequacy theorem states that if we prove $\{↯(\varepsilon)\}\ e\ \{\phi\}$ in Eris, for any meta-logic post-condition ϕ, then the final distribution obtained from running e does *not* satisfy ϕ with at most probability ε. It also states that e is safe with probability at least $1 - \varepsilon$.

As mentioned in §1, Eris is a partial correctness logic, which means that a diverging program satisfies any specification, and diverging traces of e will also be considered correct. Hence ε is an *upper bound* on the probability of terminating and not satisfying ϕ. Later on, we will present Eris$_t$, a total correctness version of Eris, which allows us to establish lower bounds on the probability of terminating and satisfying ϕ.

Although our definition of Hoare triples is new, the program logic rules for the deterministic fragment of $\lambda^{\mathrm{rand}}_{\mathrm{ref}}$ are essentially standard, *e.g.*,

$$\textsc{ht-frame}\quad \frac{\vdash \{P\}\ e\ \{Q\}}{\vdash \{P * R\}\ e\ \{Q * R\}} \qquad \textsc{ht-bind}\quad \frac{\vdash \{P\}\ e\ \{v.Q\} \qquad \vdash \forall v.\ \{Q\}\ K[v]\ \{R\}}{\vdash \{P\}\ K[e]\ \{R\}} \qquad \textsc{ht-load}\quad \frac{\ell \mapsto w \vdash P(w)}{\vdash \{\ell \mapsto w\}\ !\ell\ \{v.\ P(v)\}}$$

$$\textsc{ht-rec}\quad \frac{\forall w.\ \{P\}\ (\mathsf{rec}\ f\ x = e)\ w\ \{Q\} \vdash \{P\}\ e[v/x][(\mathsf{rec}\ f\ x = e)/f]\ \{Q\}}{\vdash \{P\}\ (\mathsf{rec}\ f\ x = e)\ v\ \{Q\}}$$

Note in particular that Eris includes the standard rule ht-rec for reasoning about recursive functions.

We do not have a specialized rule for composing errors for composite computations as in previous works. The error is just a resource which, as mentioned in §1, allows us to derive an aHL-style composition rule as well as the value-dependent composition rule shown in §1 from the resource rules of credits and the ht-bind and ht-frame rules. These derived rules are shown below.

$$\frac{\vdash \{P * ↯(\varepsilon_1)\}\ e\ \{v.Q\} \qquad \vdash \forall v.\ \{Q * ↯(\varepsilon_2)\}\ K[v]\ \{R\}}{\vdash \{P * ↯(\varepsilon_1 + \varepsilon_2)\}\ K[e]\ \{R\}}\ \textsc{ht-bind-simpl}$$

$$\frac{\vdash \{P * ↯(\varepsilon_1)\}\ e_1\ \{x.\ ↯(\mathcal{E}_2(x)) * Q\} \qquad \vdash \forall x.\ \{Q * ↯(\mathcal{E}_2(x))\}\ e_2\ \{R\}}{\vdash \{P * ↯(\varepsilon_1)\}\ \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\ \{R\}}\ \textsc{ht-bind-exp}$$

Proof rules for sampling. The only rules that make direct use of error credits are our novel rules involving sampling. As mentioned in §1, Eris includes a general rule for sampling, which takes the *expected* number of error credits into account. When taking a probabilistic step, this allows us to make the error depend on the result of the step. Suppose that we sample i uniformly from the set $\{0, \ldots, N\}$ and, moreover, that we know that any continuation needs $\mathcal{E}_2(i)$ credits, where $\mathcal{E}_2 : \{0, \ldots, N\} \to [0, 1]$. Then it suffices to have $\mathbb{E}[\mathcal{E}_2]$ at the start, since it is the expected number of error credits that our proof will need. We can also interpret this principle forwards: whenever we take a step, we can split our ε error credits across all possible branches as a weighted sum according to the probability of each branch. Formally, this is captured in the following rule:

$$\frac{\sum_{i=0}^{N} \dfrac{\mathcal{E}_2(i)}{N+1} = \varepsilon_1}{\vdash \{↯(\varepsilon_1)\}\ \mathsf{rand}\ N\ \{n\ .\ ↯(\mathcal{E}_2(n))\}}\ \textsc{ht-rand-exp}$$

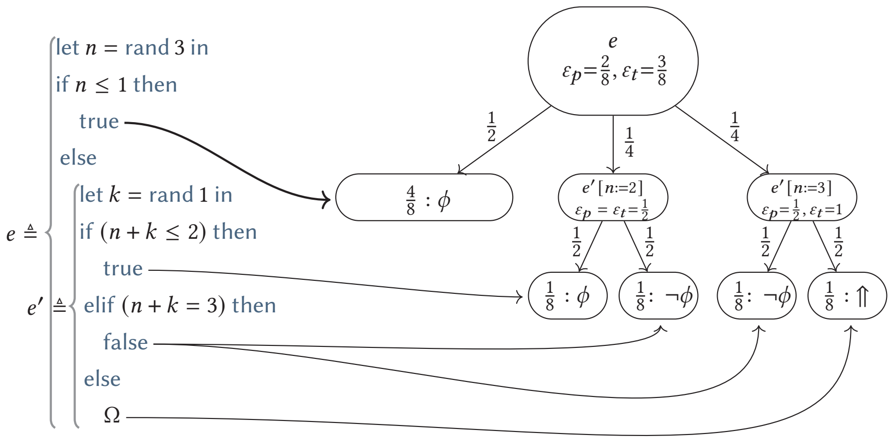

Fig. 1. Expected error analysis, partial and total.

In practice, we often want to avoid the generality of this rule as it requires explicit accounting of error credits for every possible outcome. We derive rules for applications that only require a more coarse-grained analysis. For instance, when sampling from $\{0, \ldots, N\}$, we can guarantee that the result of our sampling will not be in a list of error values xs by using the following derived rule:

$$\frac{}{\vdash \{\not{z}(\mathsf{length}(xs)/(N+1))\}\ \mathsf{rand}\, N\ \{n\,.\, n \notin xs\}}\ \textsc{ht-rand-err-list}$$

In order to derive the rule, it suffices to apply HT-RAND-EXP using

$$\mathcal{E}_2(n) \triangleq \mathsf{if}\ (n \in xs)\ \mathsf{then}\ 1\ \mathsf{else}\ 0$$

and immediately discard all outcomes in xs using $\not{z}(1) \vdash \bot$.

Example. Consider the program e and its probabilistic execution tree shown in Figure 1. The program first samples a natural n at random from 0 to 3, returning true if it chooses 0 or 1. Otherwise the sub-program e' samples a random bit k and branches on the sum of $n + k$: when $n + k \leq 2$ the program returns true, when $n + k = 3$ the program returns false, and otherwise the program diverges (denoted by Ω).

We will now show how to use Eris to show that e returns true up to some error bound. Before we present the proof, let us take a step back and ask ourselves: What specification can we hope to show for e? If we consider all the possible executions of e, we can determine that it returns a value that satisfies ϕ with probability $\frac{5}{8}$, returns a value that does *not* satisfy ϕ with probability $\frac{1}{4}$, and loops forever with probability $\frac{1}{8}$. Therefore, we should be able to prove the Hoare triple:

$$\left\{\not{z}\left(\tfrac{1}{4}\right)\right\}\ e\ \{\phi\}$$

To prove this triple, we first prove a triple for the subexpression e':

$$\left\{(n = 3 \vee n = 2) * \not{z}\left(\tfrac{1}{2}\right)\right\}\ e'\ \{\phi\}$$

This Hoare triple is not difficult to prove; during the rand 1 step, we use the HT-RAND-ERR-LIST rule to spend $\not{z}\left(\frac{1}{2}\right)$ and "avoid" values that eventually lead to undesirable outcomes, e.g. returning false. To be more specific, for the case where $n = 2$, we avoid sampling 1 as that branch eventually reduces to false. Similarly, for the case where $n = 3$, we avoid sampling 0. After verifying the sub-program e', we can now turn to verifying the overall program e. Notice that after assigning a random value to n, the number of error credits that we need for the continuation depends on n. This dependency is captured by the $\mathcal{E}_2$ function defined below:

$$\mathcal{E}_2(n) = \begin{cases} 0 & x < 2 \\ \frac{1}{2} & x = 2 \lor x = 3 \end{cases}$$

Since $\sum \mathcal{E}_2(i)/4 = 1/4$ we apply the HT-RAND-EXP rule to conclude $\left\{ ↯ \left(\frac{1}{4}\right) \right\}$ rand 3 $\{n. \mathcal{E}_2(n)\}$ and by using HT-BIND-EXP we complete the proof.

To summarize, this example demonstrates how we can use the proof rules of Eris to distribute error credits across many branches in a fine-grained manner and establish a strict expected error bound. Moreover, the error analysis in the proof is modular in the sense that we prove the properties of the sub-program e' without taking into account the context in which it is used.

4 Case Studies

In this section we present a series of case studies that showcase the features and reasoning principles introduced by Eris. Due to the representation of errors as a resource, Eris allows us to do precise, value-dependent reasoning about error bounds for operations on randomized data structures. However, when using these randomized data structures as a client, value dependent specifications often reveal too many details about the internal state of the data structure and its implementation. Here we also show how to do *amortized error* reasoning which, analogously to bounds on amortized running time, assigns a uniform error cost to every operation on the data structure despite the real error cost varying over time.

4.1 Dynamic Vectors under a Faulty Allocator

A quintessential example of amortized *time complexity* reasoning is that of a vector with dynamic resizing, see [Cormen et al. 2009]. We assume that on initialization we will allocate a memory block of size N, which allows us to do N insertions (each of cost 1) into the vector. For the $(N + 1)$-th insertion however, we allocate a new block of memory of size $2N$ and copy the contents of the vector into the new block. This operation incurs a cost of $N + 1$, paying N to copy the initial N elements into the new memory block, and 1 for the actual insertion. Using amortized cost reasoning we can argue that each insertion has amortized cost 3: 1 for the insertion itself, 1 to pay for the first time it gets copied, and 1 to pay to copy another element that was inserted and moved previously.

We will use a similar intuition to introduce amortized *error* reasoning. Here we consider a faulty memory allocator which has a small probability ε of failing on each write operation. Specifically, we assume that the allocator offers two methods extend and store with specifications:

$$\{↯(n \cdot \varepsilon) \ * \ l \mapsto^* vs\} \ \mathsf{extend}\ n\ l \ \{l'. l' \mapsto^* (vs ++ \mathsf{replicate}(n, ()))\}$$
$$\{↯(\varepsilon) \ * \ l \mapsto^* vs * n < \mathsf{length}(vs)\} \ \mathsf{store}\ l\ n\ v \ \{l \mapsto^* (vs[n := v])\}$$

Here, $\mapsto^* vs$ denotes ownership of a points-to connective for each element of the list vs. Provided we own $l \mapsto^* vs$, we can get a new, extended memory block starting at a new location l' containing the old array with n new empty locations (containing ()) appended. This incurs an error cost of $↯(n \cdot \varepsilon)$. We can also use the store operation to write to any position n within vs and update its value, again by incurring an error cost of $↯(\varepsilon)$. Consider now the following code for the pushback method which adds an element v to the end of the vector vec. It is parametrized by two methods ext and str for extending and storing:

$$\begin{aligned}\text{pushback ext str vec } v \triangleq\ & \textsf{let } (l, s, r) = \text{vec } \textsf{in} \\ & \text{str } l\ s\ v; \\ & \textsf{if } s + 1 == r \textsf{ then } (\text{ext } r\ l, s + 1, 2 \cdot r) \\ & \qquad\qquad \textsf{else } (l, s + 1, r)\end{aligned}$$

A vector is a tuple (l, s, r) of a location l pointing to the start of the vector and two integers s, r denoting the current size of the vector and the current size of the allocated block, respectively. On insertion we store value v at position s, and if we reach the end of the current allocated space we resize it so that the new block has size $2 \cdot r$.

The representation predicate for the vector looks as follows:

$$\begin{aligned}\mathit{vec\_spec}\ \text{vec vs} \triangleq \exists l, s, r, xs, p.\ vs = (l, s, r) * \lightning(p) * l \mapsto^* (vs +\!\!+\ xs)\ * \\ s < r * s = \text{length}(vs) * r = \text{length}(vs) + \text{length}(xs)\ * \\ p + 2 \cdot \varepsilon \cdot \text{length}(xs) = r \cdot \varepsilon\end{aligned}$$

Here, *vec_spec* vec vs should be read as "vec is a vector containing the values vs". Internally, the representation predicate contains the starting location of the vector l, its current size s, the size of the allocated space r and a list of dummy values xs. Crucially, it also stores a reserve of p error credits. We also know that there are $\text{length}(xs)$ insertions remaining until resizing, and on each we will leave 2ε credits to spare. Altogether, this suffices until the next resizing, which has cost $r \cdot \varepsilon$.

With this representation predicate, we can prove the following specification for pushback.

$$\{\mathit{vec\_spec}\ \text{vec vs} * \lightning(3 \cdot \varepsilon)\}\ \text{pushback extend store vec v}\ \{\text{vec}', \mathit{vec\_spec}\ \text{vec}'\ \text{vs} +\!\!+ [\text{v}]\}$$

Ignoring the error credits for the moment, this is a natural specification: if we have a vector containing vs and we append v, we get a vector containing vs $+\!\!+$ [v]. We just give a quick sketch of the proof, focusing on the accounting of credits, as the rest is standard separation logic reasoning. First, we split $\lightning(3 \cdot \varepsilon)$ into $\lightning(\varepsilon) * \lightning(2 \cdot \varepsilon)$, using the first ε credits to pay for the call to store. From the definition of the representation predicate we get $\lightning(p)$, and we can split the proof into two cases depending on whether $s + 1 < r$ or $s + 1 = r$. In the first case, which steps into the else branch of the conditional, we store back $\lightning(2 \cdot \varepsilon + p)$ into the representation predicate. It is easy to see that

$$p + 2 \cdot \varepsilon + 2 \cdot \varepsilon \cdot (\text{length}(xs) - 1) = r \cdot \varepsilon$$

since we will overwrite the first dummy location of xs. In the second case, we step into the then branch to resize. Here we know from the representation predicate that $\text{length}(xs) = 1$ since there is only one dummy location left to be overwritten. Therefore, the representation predicate implies

$$p + 2 \cdot \varepsilon = r \cdot \varepsilon$$

We own exactly $\lightning(p) * \lightning(2 \cdot \varepsilon)$, which we use to pay for the extend operation with cost $\lightning(r \cdot \varepsilon)$. At the end, we store 0 error credits into the representation predicate. This completes the proof.

4.2 Amortized Error for Collision-Free Hash Functions

We now implement a model of an idealized hash function under the *uniform hash assumption* [Bellare and Rogaway 1993], *i.e.*, a hash function h from a set of keys K to values V that behaves as if, for each key k, the hash $h(K)$ is randomly sampled from a uniform distribution over V independently of all other keys. We implement the model using a mutable map lm, which serves as a cache of hashes computed so far. If the key k has already been hashed we return the value stored in $lm(k)$. Otherwise, we sample a fresh value uniformly from $V = \{0, \ldots, n\}$, store it in $lm(k)$, and return it.

$$\begin{array}{rl} \mathsf{compute\_hash}\ lm\ v \triangleq & \mathsf{match}\ \mathsf{get}\ lm\ v\ \mathsf{with} \\ & \quad \mathsf{Some}(b) \Rightarrow b \\ & \mid \mathsf{None} \quad\ \Rightarrow \mathsf{let}\ b = \mathsf{rand}\ n\ \mathsf{in} \\ & \qquad\qquad\qquad \mathsf{set}\ lm\ v\ b; \\ & \qquad\qquad\qquad b \\ & \mathsf{end} \end{array}$$

To reason about the correctness of many data structures, we often assume that a hash function is *collision-free* in the sense that for the finite number of times we query the hash function, different input keys will return different hash values. In reality collisions may occur, but when the size of V is magnitudes larger than the number of times we use the hash function it is common to postulate that the hash function will remain collision-free, up to some small error.

To be precise, suppose we have queried $hfun \triangleq \mathsf{compute\_hash}\ lm$ a total of s times, each with a distinct input, and that the map is still collision-free (that is, we have observed s different values). If we apply the hash function to a completely new input, in order to maintain the collision-free property the hash function needs to "avoid" sampling any of the previous s hash outputs. We can reason about this by means of the HT-RAND-ERR-LIST rule, meaning we would need to pay $\lightning\left(\frac{s}{n+1}\right)$ when choosing the new hash. We can encode this as a specification for our hash function in Eris:

$$\left\{n \notin \mathrm{dom}\ m * cf\_hashfun\ lm\ m\ V * \lightning\left(\frac{\mathrm{size}(m)}{n+1}\right)\right\}\ hfun\ n\ \{v.\ cf\_hashfun\ lm\ (m[n \leftarrow v])\ V\}$$

The predicate $cf\_hashfun\ lm\ m\ V$ states that the mutable map lm tracks the finite partial function $m : \mathbb{N} \rightharpoonup \{0, \ldots, V\}$ represented as a finite map, and furthermore states that m is injective (*i.e.* there are no collisions). After querying the hash function for an unhashed key n, it will return a value v and update the mutable map to track the finite map $m[n \leftarrow v]$, which is again injective.

One limitation of the above specification is that the error requirement for each hash operation is proportional to the size of the map. This leads to worse modularity, since a client of this data structure needs to know how many queries have been performed before, which may be challenging *e.g.*, in the presence of concurrency where multiple clients may share the same hash function.

One possible solution is to fix a maximum global number of hash queries MAX and amortize the error over all those queries, so that for each query, the error one needs to pay is a fixed constant that is not dependent on the inner map. As with the previous example, we will implement this using error credits.

Starting from an empty map, if we bound the number of queries to be MAX, the total number of error credits used for the MAX queries is $\sum_{i=0}^{\mathrm{MAX}-1} \frac{i}{n+1} = \frac{(\mathrm{MAX}-1)*\mathrm{MAX}}{2(n+1)}$. We will require that the client always incurs the mean error $\frac{(\mathrm{MAX}-1)*\mathrm{MAX}}{2(n+1)*\mathrm{MAX}} = \frac{(\mathrm{MAX}-1)}{2(n+1)}$. Let $\varepsilon_{\mathrm{MAX}} = \frac{(\mathrm{MAX}-1)}{2(n+1)}$. Updating our specification,

$$\left\{\begin{array}{c} \mathrm{size}(m) < \mathrm{MAX} * n \notin \mathrm{dom}\ m * \\ amort\_cf\_hashfun\ lm\ m\ V * \lightning(\varepsilon_{\mathrm{MAX}}) \end{array}\right\}\ hfun\ n\ \{v.\ amort\_cf\_hashfun\ lm\ (m[n \leftarrow v])\ V\}$$

In Eris, this new specification is *derivable* from the original non-amortized specification. We accomplish this by defining the abstract predicate *amort_cf_hashfun* to not only contain the *cf_hashfun* resource, but also a reserve of extra error credits which the clients paid in excess for the first half of the hash operations (similar to the dynamic vector example). For the second half of the hash operations, when the mean error $\varepsilon_{\mathrm{MAX}}$ is insufficient to apply the original specification, we draw the additional error credits from the reserve in *amort_cf_hashfun*. By using error credits, we provide a simpler interface to our initial specification which alleviates the error accounting burden from clients of *amort_cf_hashfun*.

4.3 Collision-Free Resizing Hash Functions

We can go one step further and implement a collision-free hash function with constant amortized insertion error, but without imposing any *a priori* limit on the number of insertions. Of course with a fixed set V of possible hash values (as in the implementation above), collisions are eventually unavoidable. Instead, we will keep the probability of collision low by resizing the *sample space* once a threshold of inserted elements is reached. One way to think of this model is to assume that the hash function gets values over a much larger sample space, but initially we only look at the first n, and every time we resize we look at the $(n+1)$-th bit.

As in the previous example, the hash is sampled lazily. In addition to a mutable map m the hash function will keep track of three quantities V, S, and R. Here V represents the size of the value space of our hash function, which are nonnegative integers over $\{0, \dots, V\}$. The value S represents the current size of the domain of the hash function, and R represents a threshold on the amount of stored values after which the hash will resize. That is, once S reaches R, we will update the hash so that R becomes $2 \cdot R$ and V becomes $2 \cdot V$. Initially V, S, R are set to some default values $V_0, 0, R_0$. We will prove that overall, the hash will remain collision-free with an amortized error of $(3 \cdot R_0)/(4 \cdot V_0)$ per insertion, *no matter the number of insertions*.[4] For instance, to keep the amortized error below 2^{10} we can initially set $V_0 = 2^{10} \cdot R_0$.

The code for querying the hash function is shown below:

```
hash_rs hf w ≜ let (lm, v, s, r) = hf in
                 match get lm w with
                   Some(b) ⇒ (b, hf)
                 | None    ⇒ let b = rand (v − 1) in
                              set lm w b;
                              if s + 1 = r then (b, (lm, 2 · v, s + 1, 2 · r))
                              else (b, (lm, v, s + 1, r))
                 end
```

Note that, the code is analogous to the non-resizing hash besides tracking the size: in the case where $s + 1 = r$ we double the value of both v and r. The specification uses the following predicate:

$$
\begin{aligned}
\mathit{cf\_hash\_rs}\ hf\ m\ v\ s\ r \triangleq \exists lm, p.\ hf = (lm, v, s, r) * ↯(p) && (1)\\
p + (\mathit{rval} - \mathit{sval}) \cdot ((3 \cdot R_0)/(4 \cdot V_0)) \geq \textstyle\sum_{i=s}^{r-1} i/v\ * && (2)\\
\mathit{cf\_hashfun}\ lm\ m\ v\ * (\dots) && (3)
\end{aligned}
$$

We explain the representation predicate line by line. The first line contains the internal representation of the hash function as a tuple, and a reserve of p error credits. The second line imposes a condition on p, namely that the current number of credits in the reserve plus the credits we will get until the next resizing is enough to pay for all of the error of the insertions until the next resizing. Note that when there are s elements in the image of the hash function and we sample uniformly over $\{0, \dots, v-1\}$, the error we will have to pay is $↯(s/v)$. The third line states that lm points to a list that represents the partial map m, and that there are no collisions. Finally, the rest of the predicate contains some constraints on the sizes v, s, r that we omit for brevity. In this specification

[4]Of course, even with this constant error cost, if we execute a large enough number of insertions we will eventually have consumed over 1 error credit. The advantage of this specification is that it will enables us to do more modular proofs, since the cost will be constant independently of the internal state of the hash function.

we have decided to expose v, s and r to the client as we will use those values in the next section, however it is also possible to hide these values from the client when those details are not needed.

We prove three specifications for hash_rs, depending on the initial conditions. If we query an element that was already in the domain of the hash function, we just get back its hash value, without the need for spending error credits:

$$\left\{ \begin{array}{c} m[w] = \mathsf{Some}\ b\ * \\ cf\_hash\_rs\ f\ m\ v\ s\ r \end{array} \right\} \mathsf{hash\_rs}\ f\ w \left\{ (b', f').\ \begin{array}{l} b' = b\ * \\ cf\_hash\_rs\ f'\ m\ v\ s\ r \end{array} \right\}$$

If we query for an element that is not in the domain we will have to sample it in a collision-free manner, at a cost of $\lightning(s/v)$. From the precondition we have $\lightning((3 \cdot r_0)/(4 \cdot v_0))$, and from the representation predicate we can get the reserve $\lightning(p)$. We can derive that this is enough to pay for $\lightning(s/v)$ from condition (2) by an uninteresting, albeit nontrivial, calculation. Since the code branches depending on whether $s + 1$ is equal to r, we have the following specification:

$$\begin{array}{l} \left\{ m[w] = \mathsf{None}\ *\ (s + 1 \leq r)\ *\ cf\_hash\_rs\ f\ m\ v\ s\ r\ *\ \lightning\left(\frac{3r_0}{4v_0}\right) \right\} \\ \quad \mathsf{hash\_rs}\ f\ w \\ \left\{ (b, f').\ \begin{array}{l} (s + 1 < r * cf\_hash\_rs\ f'\ m[w := b]\ v\ (s + 1)\ r) \vee \\ (s + 1 = r * cf\_hash\_rs\ f'\ m[w := b]\ (2 \cdot v)\ (s + 1)\ (2 \cdot r)) \end{array} \right\} \end{array}$$

In both cases we will have to reestablish the representation predicate. In particular, we will have to store the remaining credits back into the reserve, and prove that condition (2) is still valid (*i.e.*, that we will have enough credits to pay for future insertions). This follows again by arithmetic calculations.

4.4 Collision-Free Resizing Hash Map

Hash maps (or hash tables) are one of the most ubiquitous data structures in programming, since they can represent large sets with efficient insertion, deletion, and lookup. Their efficiency relies on having a low number of collisions, so that each location on the table contains a small number of values. As the number of collisions increases, and thus the performance of the hash map worsens, it is often beneficial to resize the table, redistributing the hashed values and freeing up space for new insertions.

In order to be able to reason about the efficiency of hash maps, we need to compute the probability of a hash collision. Computing this probability over a sequence of insertions can be cumbersome, as it depends on the current size of the hash map and the number of elements it contains. As a consequence, it can lead to less modular specifications for programs that use hash maps inside their components.

We will use the dynamically-resizing hash function defined above to implement a collision-free dynamically-resizing hash map, and specify it with an amortized cost for insertion. Namely we will use an array of size v, in which s entries are filled with a hashed value and the rest are uninitialized. Once we fill in r elements, we resize the table to have size $2v$ and we set r to $2r$. New hash elements are sampled in a collision-free manner following the specification shown in the previous sections, thus ensuring that the hash map is also collision-free. We can then prove the following specification[5] for inserting a value w into a hash map hm:

$$\{\mathsf{isHashmap}\ hm\ ns\ *\ \lightning((3 \cdot R_0)/(4 \cdot V_0))\}\ insert\ hm\ w\ \{hm'.\ \mathsf{isHashmap}\ hm'\ (ns \cup \{w\})\}$$

Recall here that V_0 is the initial capacity of the hash table, and R_0 is the threshold on which we will first resize. The representation predicate isHashmap hm ns should be understood as "hm is

[5]The code for this example and the definition of the representation predicate can be found in Appendix B

a collision-free hash map representing the set (of natural numbers) *ns*". Crucially, this predicate does not keep track of error credits as all of the error accounting is done through the *cf_hash_rs* predicate, which is used as a client within isHashmap. This specification states that an insertion of an element w fail with probability at most $(3 \cdot R_0)/(4 \cdot V_0)$. There are two cases in proving this specification: either w was already in the hash map (and therefore $ns = ns \cup \{w\}$) or it is a new element. The former case is immediate; if it is a new element, we can use $↯((3 \cdot R_0)/(4 \cdot V_0))$ to sample a fresh value from the hash function using the specifications proven in the previous section. This ensures that the location in the table corresponding to that index is uninitialized. Since the hash map resizes at the same time as the hash function does, this establishes our specification no matter how many insertions have been performed before.

4.5 Amortized Hash Functions and Merkle Trees

A *Merkle tree* [Merkle 1987] is a data structure that relies on a hash function. It is used to ensure the authenticity and validity of data received from a potentially unreliable and malicious source and used widely in, *e.g.*, distributed file systems [Benet 2014] and databases [DeCandia et al. 2007].

A Merkle tree is a binary tree whose nodes contain pairs consisting of a value and a label. For leaves, the label is the hash of the value stored in the leaf. In the case of inner nodes, the label corresponds to the hash of the concatenation of the labels of its children. We call the label of the root of a Merkle tree a *root hash.* Merkle trees are interesting because they support constructing a cryptographic *proof certificate* that a value is in a leaf of the tree. These proofs can be validated by a client who only knows the root hash of the tree.

To construct a proof that a value v is in the tree, we start from the leaf l containing the value v, The proof starts with the hash of the *sibling* of l. We then traverse up from the leaf l to the root along the ancestors of l, appending to the proof the hash annotations of the *siblings* of each ancestor we traverse. A client who has the root hash h can check the proof by effectively computing a list fold over the proof, successively hashing each element of the proof against an accumulated hash. The client then checks whether the result of the fold matches the root hash h; if it matches, the proof is deemed valid, and otherwise it is rejected as invalid.

Why is this proof checking procedure sound? For an invalid proof to be (incorrectly) validated by a checker, the invalid proof must contain values that cause a colliding hash value to be computed during the checker's list fold. Thus, if an adversary cannot find a collision, they cannot maliciously convince a checker with an invalid proof. In particular, if the hash is treated as a uniform random function, and the total number of distinct hashes ever computed is relatively small (*e.g.*, because of constraints on the adversary's computational power), the probability that the proof checking procedure will accept an invalid proof is very small. In this example, we prove such an error bound under the assumption of a bound on the total number of hashes ever computed.

We use the fixed-size amortized hash with values in $0, \ldots, 2^{V-1}$ to implement a library for Merkle trees. Given a possible leaf value and a purported proof, together with an error credit of $↯(\varepsilon_{\mathsf{MAX}} \cdot \mathit{height}(\mathit{tree}))$ in the precondition, the specification for the proof checker will ensure that when a proof is invalid the checker will return false (*i.e.*, the checker is sound up to this probability of error). The amortization of the hash simplifies the specification of the checker since it incurs a constant amount of error credits which only depends on the amortized error $\varepsilon_{\mathsf{MAX}}$ and the tree but *not* the size of the map in the hash.

The checker function as shown in Figure 2 is implemented using the hash_path helper function, which recursively computes the potential root hash from the input proof and leaf value. We represent a proof as a list of tuples following the path from the leaf to the root: each tuple consists of a boolean flag to determine which child of the current node is on the path to the leaf, and the hash of the child node that is not on the path. In the base case where the proof is an empty list,

```
hash_path f lproof lleaf ≜
  match lproof with
   (hd :: tl) ⇒ let (b, hash) = hd in
                if b then f ((hash_path f tl lleaf) ∗ 2^V + hash)
                 else f (hash ∗ 2^V + (hash_path f tl lleaf))
  | nil       ⇒ f lleaf
  end

checker root_hash f ≜
  λlproof, lleaf.
    let hp = hash_path f lproof lleaf in
    root_hash == hp
```

Fig. 2. A proof checker for Merkle trees.

we arrive at the leaf of the Merkle tree, and we return the hash of our input leaf value. In the intermediate step, where we arrive at a branch of the tree, we recursively compute the potential hash value of the branch containing the leaf node and bit-wise concatenate it with the hash found in the head element of our proof. We then return the hash of this concatenated number.

Our simplified specification for the checker is displayed below:

$$\left\{\begin{array}{c} \mathit{isList}\ l\ \mathsf{lproof}\ * \\ \mathit{tree\_valid}\ \mathit{tree}\ m\ * \\ \mathit{amort\_cf\_hashfun}\ \mathsf{f}\ m\ * \\ \mathit{size}(m) + \mathit{height}(\mathit{tree})\ <=\ \mathsf{MAX}\ * \\ \lightning(\varepsilon_{\mathsf{MAX}} \cdot \mathit{height}(\mathit{tree})) \end{array}\right\}$$

$$\mathsf{checker\ root\_hash\ f\ lproof\ v} \quad \left\{ b.\ \begin{array}{l} \text{if } b \\ \text{then } \mathit{tree\_leaf\_proof\_match}\ \mathit{tree}\ \mathsf{v}\ \mathsf{lproof}\ *\ \ldots \\ \text{else } \mathit{not\_tree\_leaf\_proof\_match}\ \mathit{tree}\ \mathsf{v}\ \mathsf{lproof}\ *\ \ldots \end{array} \right\}$$

Line-by-line, the precondition here says that:

(1) the value lproof is represented by the abstract mathematical list l,
(2) the Merkle tree *tree* is built correctly according to the hash map m,
(3) the function f encodes the amortized hash function under the map m,
(4) the size of m plus the height of the tree is smaller or equal to MAX,
(5) we have credits equal to amortized error multiplied by the height of the tree.

The postcondition states that the Boolean returned by checker soundly represents the inclusion of v in the tree. We impose the inequality of the size of the map m in the beginning since checker runs the hash function exactly $\mathit{height}(\mathit{tree})$ times (recall that there is a total limit on the number of distinct hashes that can be computed).

How are the error credits used to derive this specification? As long as the hash function remains collision-free throughout the checking procedure then any corrupted data will modify all hashes above it in the tree—in particular, it will change the root hash. Therefore, we will spend error credits at each of the $\mathit{height}(\mathit{tree})$ hashes computed by checker to preserve collision-freedom

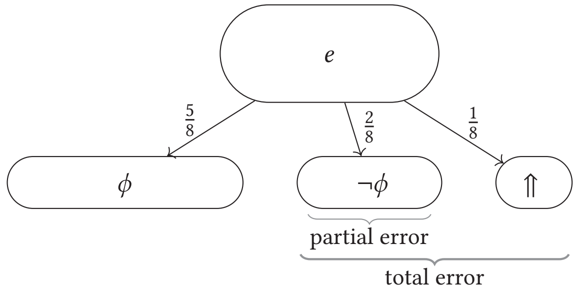

Fig. 3. Partial and total approximate correctness contrasted.

throughout the checking process. We remark that if we chose to use a non-amortized hash for the implementation of the Merkle tree library, the amount of error credits paid as one traverses the tree may change if a new value is ever encountered, leading to a more convoluted specification.

4.6 Further Case Studies

For reasons of space, we omit other case studies, which can be found in the long version of the paper. In particular, we include an example that uses the Merkle tree as a client to store data into an unreliable storage system and prove that, with high probability, it can be used to detect data corruption.

5 Almost-Sure Termination via Error Credits

In this section, we introduce Eris$_t$, an *approximate total-correctness* version of Eris, and show how it can be used to prove almost-sure termination via reasoning about error credits. Before we embark on the technical development, we consider for a second time the example from Figure 1, which illustrates the distinction between partial and total correctness interpretations of approximate reasoning up to some error. A summary of the example is depicted in Figure 3.

In §3 we showed that for the program e in Figure 1, we can show the specification Hoare triple $\left\{ ↯(\varepsilon_p) \right\}\ e\ \{x.\, x = \mathsf{true}\}$ for $\varepsilon_p = \frac{2}{8} = \frac{1}{4}$ in Eris, intuitively because the program terminates with a result not satisfying the postcondition (false) with probability $\frac{1}{4}$. Note that the probability of non-termination ($\frac{1}{8}$) is not included in the error, since non-termination is considered acceptable by the partial-correctness interpretation of Eris.

In Eris$_t$ on the other hand, one cannot satisfy a Hoare triple by not terminating, and thus $\varepsilon_t = \frac{3}{8}$ is needed to show an approximate *total* Hoare triple $\langle ↯(\varepsilon_t) \rangle\ e\ \langle x.\, x = \mathsf{true} \rangle$ (we use angle brackets when we want to emphasize that we are stating a Hoare triple in Eris$_t$). The proof of this Hoare triple in Eris$_t$ is very similar to the Eris proof for e in §3, only changing the distribution of error credits at each sample so that our additional starting credit can discharge the nonterminating branch (specifically, $\varepsilon_t = 1$ in the rightmost branch of Figure 1).

Stepping back from the example, we can make a simple but crucial observation: if the total error necessary for showing a Hoare triple $\langle ↯(\varepsilon) \rangle\ e\ \langle P \rangle$ (for any postcondition P) can be proven for any arbitrary (but positive) ε, then we can make the probability of divergence vanishingly small, and e must be almost-surely terminating.

In this section, we show how to make this argument precise and then demonstrate how it yields an approach for showing almost-sure termination via error credits, which allows us to prove correctness of Las Vegas algorithms.

5.1 The Eris$_t$ Logic, Adequacy, and Almost-Sure Termination

It is important to note that all the proof rules of Eris shown earlier are still sound for Eris$_t$, with the exception of HT-REC. Note in particular that HT-REC could be applied to the diverging function $\mathsf{rec}\ f\ x = f\ x$, which is unsound in the total-correctness setting. To reason about recursive programs in Eris$_t$ we can instead use a novel technique we call *induction on the error amplification*. We will see examples below of how this principle works.

The meaning of a Hoare triple in Eris$_t$ is given by its adequacy theorem, which states that given a program e, if from the assumption $\text{↯}(\varepsilon)$ we can prove a metalogical postcondition ϕ, then the program will terminate and satisfy ϕ with at least probability $1 - \varepsilon$:

Theorem 7 (Total Adequacy). *If* $\vdash \langle \text{↯}(\varepsilon) \rangle\ e\ \langle \phi \rangle$ *then* $\Pr_{\mathrm{exec}\,(e,\sigma)}[\phi] \geq 1 - \varepsilon$ *for any state* σ.

Since the logic is total this also implies that the probability of the program crashing is bounded from above by ε. By a continuity argument in the meta logic (outlined in §6), we then obtain the following theorem.

Theorem 8 (Almost-Sure Termination). *Let* $\varepsilon \geq 0$. *If for all* $\varepsilon' > \varepsilon$, $\vdash \langle \text{↯}(\varepsilon') \rangle\ e\ \langle \phi \rangle$ *then* $\Pr_{\mathrm{exec}\,(e,\sigma)}[\phi] \geq 1 - \varepsilon$ *for any state* σ.

If we pick $\varepsilon = 0$, this theorem allows us to conclude almost-sure termination of e by proving $\vdash \langle \text{↯}(\varepsilon') \rangle\ e\ \langle \phi \rangle$ for any $\varepsilon' > 0$. Note that we here quantify over ε' in the meta-logic; the Eris$_t$ portion of this argument freely assumes ownership over some arbitrary positive $\text{↯}(\varepsilon')$.

5.2 Case Studies

We now present case studies of how we can prove almost-sure termination via error credits. In the first case study we demonstrate how we can prove a Hoare triple in Eris$_t$ by a form of induction on error credits. The second case study presents a technique to upper-bound the probability of excessively long runs of a program. In the third case study we present a novel *planner* proof rule, which can separate credit arithmetic from concrete program steps, and we show how to use it to prove correctness of general rejection samplers. Finally, we revisit an example due to McIver et al. [2018], and show how error credits can be used to provide a simple proof that it is almost surely terminating.

5.2.1 Induction by Error Amplification. A variety of *Las Vegas* algorithms employ a "sample and retry" approach, whereby a *sampler* program produces a possibly undesirable value and a *checker* program forces the sampler to retry until it produces an acceptable value. In a partial-correctness logic, it is trivial to prove that such algorithms return acceptable values when they terminate (using HT-REC in Eris). However, it is more challenging to bound the probability that they fail to terminate.

Rejection samplers are one common example of this kind of algorithm. Rejection samplers simulate complex probability distributions using sequences of samples from simpler distributions, by strategically rejecting sequences which do not correspond to values in the target distribution. Consider the implementation of a typical rejection sampling scheme ($\mathsf{RSamp}\ s\ c$) with sampler program s and checker program c below.

$$\mathsf{RSamp} \triangleq \lambda \mathsf{s}.\, \lambda \mathsf{c}.\, \mathsf{let}\ \mathsf{try} = (\mathsf{rec}\ \mathsf{try}\ \_ = \mathsf{let}\ \mathsf{v} = \mathsf{s}\ ()\ \mathsf{in}\ \mathsf{if}\ (\mathsf{c}\ \mathsf{v})\ \mathsf{then}\ \mathsf{v}\ \mathsf{else}\ \mathsf{try}\ ())\ \mathsf{in}\ \mathsf{try}\ ()$$

As an archetypical example, we can emulate samples from $\{0, 1, \ldots, N\}$ using a uniform sampler of size $M > N \geq 1$ by providing the sampler $\mathsf{us}_M \triangleq (\lambda\_.\ \mathsf{rand}\ M)$ and checker $\mathsf{uc}_N \triangleq (\lambda \mathsf{v}.\ \mathsf{v} \leq N)$.

Let us show that this uniform rejection sampler ($\mathsf{RSamp}\ \mathsf{us}_M\ \mathsf{uc}_N$) terminates almost surely. Using Theorem 8, it suffices to show, for arbitrary nonnegative ε,

$$\langle \text{↯}(\varepsilon) \rangle\ \mathsf{RSamp}\ \mathsf{us}_M\ \mathsf{uc}_N\ \langle \mathsf{True} \rangle. \tag{4}$$

In proving this, we need to reason about the recursion in RSamp. Since we no longer have HT-REC, we will have to use some form of induction, yet there appears to be no argument to induct on. The solution is a technique we call *induction by error amplification.* We first show how the principle works in detail, then derive rules that makes its use more practical. Note that using expectation-preserving composition, for any ε' we are able to prove

$$\langle \text{↯}(\varepsilon') \rangle \ \mathsf{us}_M \ \langle v. \text{↯}(E(v) \cdot \varepsilon') \rangle \qquad \text{where} \quad E(v) \triangleq \begin{cases} 0 & 0 \leq v \leq N \\ \frac{M+1}{M-N} & N < v \end{cases}$$

since $\frac{1}{M+1} \sum_{i=0}^{M} E(i) = 1$. In other words, each sampling attempt either produces a value which the checker will certainly accept, or scales our error credit by a factor of $\frac{M+1}{M-N} > 1$. This means that we can grow our error credit geometrically in the cases where a sample does not immediately terminate, and we need only repeat this procedure $d(\varepsilon) = \lceil -\log_{(M+1)/(M-N)}(\varepsilon) \rceil$ times: either some sampling attempt will succeed, or they all fail and we will have amplified the original error to $\text{↯}(1)$, from which we obtain a proof of False. Starting with any $\varepsilon > 0$, induction over $d(\varepsilon)$ allows us to prove (4), completing the proof. While this proof technique appears to be novel among total correctness logics, under the hood it closely mirrors a standard analysis in probability theory where one shows that longer and longer traces are increasingly unlikely, and concludes by taking a limit.

This induction principle is abstractly captured by the rule below:

IND-ERR-AMP
$$\frac{\varepsilon > 0 \qquad k > 1 \qquad \forall \varepsilon'. \, (\text{↯}(k \cdot \varepsilon') \twoheadrightarrow P) * \text{↯}(\varepsilon') \vdash P}{\text{↯}(\varepsilon) \vdash P}$$

The rule requires us to (1) own a positive amount of error credits ε and (2) choose an amplification factor $k > 1$. Using the rule gives us an arbitrary initial amount of error credits $\text{↯}(\varepsilon')$ and an inductive hypothesis for which we need to pay $\text{↯}(k \cdot \varepsilon')$. Soundness of the rule is proven by induction on the number of times we need to scale ε by k until we accumulate $\text{↯}(1)$, i.e. $\lceil -\log_k(\varepsilon) \rceil$.

Note that while this principle holds for an arbitrary proposition P, it only ever makes sense to use it when reasoning about programs, because otherwise we have no way of amplifying our error credits. To reason about recursive functions in the total correctness logic, we can derive the following rule as a consequence:

HT-REC-ERR
$$\frac{\begin{gathered}\varepsilon > 0 \qquad k > 1 \\ \forall \varepsilon'. \, (\forall w. \ \langle P * \text{↯}(k \cdot \varepsilon') \rangle \ (\mathsf{rec}\ f\ x = e)\ w \ \langle Q \rangle) \ \vdash \ \langle P * \text{↯}(\varepsilon') \rangle \ e[v/x][(\mathsf{rec}\ f\ x = e)/f] \ \langle Q \rangle\end{gathered}}{\vdash \langle P * \text{↯}(\varepsilon) \rangle \ (\mathsf{rec}\ f\ x = e)\ v \ \langle Q \rangle}$$

Turning again to our example, we can prove Equation (4) in an arguably simpler manner by applying HT-REC-ERR and setting $k = \frac{M+1}{M-N}$. This reduces to proving, for an arbitrary ε',

$$\langle \text{↯}(k \cdot \varepsilon') \rangle \ \mathsf{try}\ () \ \langle \mathsf{True} \rangle \ \vdash \ \langle \text{↯}(\varepsilon') \rangle \ \mathsf{let}\ \mathsf{v} = \mathsf{us}_M \ \mathsf{in}\ \ \mathsf{if}\ (\mathsf{uc}_N\ \mathsf{v})\ \mathsf{then}\ \mathsf{v}\ \mathsf{else}\ \mathsf{try}\ () \ \langle \mathsf{True} \rangle$$

Following the credit splitting strategy above, after sampling from us_M we will either have a value v that passes the check uc_N, or we will amplify our error credits to $\text{↯}(k \cdot \varepsilon')$ and go to the recursive case, in which case we can conclude immediately by instantiating the inductive hypothesis.

5.2.2 Reasoning about Tail Bounds. Another property one may want to prove about a rejection sampler is that long runs only happen with low probability. The explicit proof of termination in the example above using induction on $d(\varepsilon)$ is implicitly proving an upper bound on the probability of a run taking longer than $d(\varepsilon)$ steps. Indeed, we can make this concrete by instrumenting the rejection sampler with a counter, and make it fail once a limit count is reached.

$$\begin{aligned}\mathsf{RSamp}_{\mathsf{bd}} \triangleq \lambda \mathsf{s}.\, \lambda \mathsf{c}.\, \lambda \mathsf{m}.\ & \mathsf{let\ rec\ \ try}\ n = \\ & \quad \mathsf{if}\ n \leq 0\ \mathsf{then\ None} \\ & \quad \mathsf{else}\ (\mathsf{let\ v} = \mathsf{s}\ ()\ \mathsf{in\ if}\ (\mathsf{c\ v})\ \mathsf{then\ Some\ v\ else\ try}\ (n-1)))\mathit{in} \\ & \mathsf{try}\ m\end{aligned}$$

We can now prove the triple below:

$$\left\langle \lightning\left(\left(\tfrac{M-N}{M+1}\right)^n\right)\right\rangle \mathsf{RSamp}_{\mathsf{bd}}\ \mathsf{us}_M\ \mathsf{uc}_N\ n\ \langle v.\, \exists w.\, v = \mathsf{Some}\ w\rangle.$$

Since the sampler will only return something of the shape Some w if it succeeds in n tries or fewer, $\left(\frac{M-N}{M+1}\right)^n$ is an upper bound on the probability of the sampler of taking more than n tries to produce a valid sample. We can prove this specification by a simple (Coq-level) induction on n. The case $n = 0$ is trivial, because we begin the proof with $\lightning(1)$. In the inductive case, after the probabilistic sampling, we will either terminate or amplify our error by $\frac{M+1}{M-N}$, as in the previous case study, and then we can apply the inductive hypothesis.

The caveat of this approach is that the reasoning about the runtime is entirely intrinsic. Our adequacy theorem does not allow us to derive from the proof of a Hoare triple a statement about the concrete runtime of our programs, but we believe that we can achieve this via an integration of (deterministic) time credits [Mével et al. 2019] into this setting.

5.2.3 Presampling Tapes. In order to extract a more general proof rule from our credit amplification argument, it is useful to separate the credit accounting steps from the symbolic execution of a program. Because the amount of error credits after an expectation-preserving composition can depend on the value sampled, this necessitates some way to express the outcome of random sampling events ahead of time. Luckily, the presampling tapes by Gregersen et al. [2024] provide this mechanism exactly. In this section we first briefly recall the semantics of $\lambda^{\text{rand}}_{\text{ref}}$ with tapes as well as the proof rules for a $\iota \hookrightarrow (N, \vec{n})$ proposition, which expresses ownership of a presampling tape. In the next section we show how to use them to get a general proof rule for credit amplification. For more details on presampling tapes, we refer the reader to *loc. cit.*, where tapes were originally introduced and used to sample in an asynchronous manner.

To introduce presampling tapes in $\lambda^{\text{rand}}_{\text{ref}}$, we extend the syntax of the language as follows.

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

In addition to the heap, a state in $\lambda^{\text{rand}}_{\text{ref}}$ with presampling also contains a map from tape labels to presampling tapes. Tapes are formally pairs $(N, \vec{n})$ of an upper bound $N \in \mathbb{N}$ and a finite sequence $\vec{n}$ of natural numbers less than or equal to N. Tape allocation via $\mathsf{tape}\, e$ returns a fresh label ι and extends the state with a new empty tape ϵ with bound N if e evaluates to $N \in \mathbb{N}$. Sampling from a tape with label ι via $\mathsf{rand}\, N\, \iota$ either *deterministically* pops the first element from the list $\vec{n}$ or uniformly samples a new integer between 0 and N.

From the point of view of the Eris$_t$ logic, a presampling tape behaves somewhat similarly to standard heap location. Like in Clutch, the proposition $\iota \hookrightarrow (N, \vec{n})$ asserts ownership of a tape labelled ι with bound N and contents $\vec{n}$. We can allocate an empty tape ϵ with a specified bound.

$$\frac{}{\vdash \langle \mathsf{True} \rangle \ \mathsf{tape}(N) \ \langle \iota. \iota \hookrightarrow (N, \epsilon) \rangle} \text{ALLOC-TAPE}$$

Sampling from a non-empty tape consumes the first value of the list.

$$\frac{}{\vdash \langle \iota \hookrightarrow (N, n \cdot \vec{n}) \rangle \ \mathsf{rand}\, N\, \iota \ \langle x. x = n * \iota \hookrightarrow (N, \vec{n}) \rangle} \text{LOAD-TAPE}$$

Note that there are no primitives in $\lambda^{\mathrm{rand}}_{\mathrm{ref}}$ for directly writing to or adding values to tapes and values are only added to tapes via *ghost operations* that appear purely at the logical level of an Eris proof.

$$\frac{\vdash \langle \iota \hookrightarrow (N, \vec{n} \cdot n) \rangle \ e \ \langle P \rangle \qquad 0 \le n \le N}{\vdash \langle \iota \hookrightarrow (N, \vec{n}) \rangle \ e \ \langle P \rangle} \text{PRESAMPLE}$$

Crucially, Eris extends the above proof rules for tapes taken from [Gregersen et al. 2024] with the following additional rule, connecting error credits in expectation to presampling via tapes.

$$\frac{\sum_{i=0}^{N} \frac{\mathcal{E}_2(i)}{N+1} = \varepsilon_1 \qquad \forall n.\ \vdash \langle \iota \hookrightarrow (N, \vec{n} \cdot n) * \lightning(\mathcal{E}_2(n)) \rangle \ e \ \langle P \rangle}{\vdash \langle \iota \hookrightarrow (N, \vec{n}) * \lightning(\varepsilon_1) \rangle \ e \ \langle P \rangle} \text{PRESAMPLE-EXP}$$

Similarly to HT-RAND-EXP, this rule allows us to sample a random integer n, write it to a tape, and get $\mathcal{E}_2(n)$ error credits, assuming that our initial amount of error credits ε_1 is the expected value of $\mathcal{E}_2(n)$. Since no execution step is taken, this allows us to disentangle credit arithmetic from the operational semantics of the program. Analogous rules also hold in the partial version of the logic, but the interaction between tapes and error credits is particularly useful for total correctness.

5.2.4 The Planner Rule. Equipped with the ability to perform credit reasoning and symbolic execution separately, we can now derive an induction principle for Eris_t which eliminates the need to perform fine-grained credit arithmetic as in §5.2.1. Our proof rule is directly inspired by the "planner" method by Arons et al. [2003]. Arons et al. establish a proof system wherein to prove termination, one can reason as if a sequence of randomized choices will yield some prover-selected sequence of outputs infinitely often (as justified by the Borel-Cantelli lemma from classical probability theory). Expressed in Eris_t, we have the following *planner* rule:

$$\frac{0 < \varepsilon \qquad \forall s.\ |z(s)| \le L \qquad \vdash \langle \exists ys.\ \iota \hookrightarrow (N, xs \mathbin{+\!\!+} ys \mathbin{+\!\!+} z(xs \mathbin{+\!\!+} ys)) \rangle \ e \ \langle \phi \rangle}{\vdash \langle \iota \hookrightarrow (N, xs) * \lightning(\varepsilon) \rangle \ e \ \langle \phi \rangle} \text{PRESAMPLE-PLANNER}$$

The rule states that if we have a tape with contents xs and any positive amount of error credits ε, then we can update our tape with some unknown sequence of "garbage" samples ys in order to ensure it includes a prover-selected target sequence $z(xs \mathbin{+\!\!+} ys)$ at the end. Generalizing Arons et al.'s planner rule, our version allows the target sequence to be a function z of the current state of the tape, provided that the length of the target word has a fixed upper bound L (and consequently, cannot become arbitrarily unlikely). After invoking the planner rule, a prover can consume the samples in $xs \mathbin{+\!\!+} ys$ using a regular induction on lists, eventually ending up in their desired tape state $\iota \hookrightarrow (N, z(xs \mathbin{+\!\!+} ys))$.

The planner rule is derivable entirely within Eris_t using the rules we have already seen. The proof proceeds by induction by error amplification, and most of the proof is directly analogous to the argument in §5.2.1. The key step lies in proving the following *amplification lemma* which, for a particular constant $K_{N,w} > 1$ (dependent on N and $|w|$), allows us to either sample a target word w

onto a tape, or sample a garbage string j and scale our error credit by a factor of $K_{N,w}$:

$$\frac{\vdash \left\langle \exists j.\, \iota \hookrightarrow (N, \vec{n} +\!\!+ j) * \not{\varepsilon}\left(K_{N,w} \cdot \varepsilon\right)\right\rangle e \,\langle \phi \rangle \qquad \vdash \langle \iota \hookrightarrow (N, \vec{n} +\!\!+ w)\rangle\, e\, \langle \phi \rangle}{\vdash \langle \iota \hookrightarrow (N, \vec{n}) * \not{\varepsilon}(\varepsilon)\rangle\, e\, \langle \phi \rangle}$$

Proving this involves $|z(\vec{n})|$ applications of PRESAMPLE-EXP. We provide the details, including the amplification constant $K_{N,w}$, in Appendix C.

Example. To demonstrate how the planner rule can eliminate the credit accounting in *induction by error amplification*, we will prove that a Poisson trial almost surely terminates. A Poisson trial is a random process, whose value comes from counting how many independent attempts a random variable takes before meeting some criteria. We can implement a Poisson trial that flips pairs of fair coins until they are both heads as an instantiation of a rejection sampler S:

$$\mathsf{cs}_l \triangleq \lambda\_.\, l \leftarrow (!\, l + 1); (\mathsf{rand}\, 1,\ \mathsf{rand}\, 1) \qquad \mathsf{cp}_l \triangleq \lambda v.\, v == (1,\ 1)$$

Note that the sampler here both maintains internal state (counting the number of trials) and uses multiple calls to rand. We seek to show

$$\langle \not{\varepsilon}(\varepsilon)\ * 0 < \varepsilon * l \mapsto 0\rangle\, S\, \mathsf{cs}_l\ \mathsf{cp}_l\ \langle \mathsf{True}\rangle.$$

Starting with any tape $\iota \hookrightarrow (N, xs)$, we invoke the planner rule with $\not{\varepsilon}(\varepsilon)$ and the target function

$$z(s) = \begin{cases} [1, 1] & |s| \text{ is even,} \\ [0, 1, 1] & \text{otherwise.} \end{cases} \tag{5}$$

This results in a tape of the form $\iota \hookrightarrow (1, xs +\!\!+ j +\!\!+ [1, 1])$ where j has even length. By induction over j, we consume the entire garbage section of the tape, with each invocation of cs_l pulling off the two samples. Either the garbage section will happen to contain some spurious $[1, 1]$ sample, in which case the program terminates, or we step through the entire garbage section and end up with tape $\iota \hookrightarrow (1, [1, 1])$ which will also cause termination. We obtain our almost-sure termination result using Theorem 8, as our initial error credit was arbitrarily small.

5.2.5 The Escaping Spline. In this case study, we revisit Example 5.4 from [McIver et al. 2018], which presents a so-called *escaping spline*. This consists in a random walk over the non-negative integers. An agent is at a position indicated by an integer n and on every step it chooses probabilistically between stopping, or moving to $n + 1$. However, the bias changes with the current position n, the probability it chooses to stop is $\frac{1}{n+1}$. Our goal is to show that, despite the probability of choosing to stop decreasing as time goes on, this walk is almost surely terminating, no matter the initial position. We can implement the walk by the following program:

$$\begin{aligned} \mathsf{rec\ spline}\ \ \mathsf{n}\ \ = \quad & \mathsf{let}\ x = \mathsf{rand}\, n\ \mathsf{in} \\ & \mathsf{if}\ x = 0\ \mathsf{then}\ ()\ \mathsf{else\ spline}\ (n + 1) \end{aligned}$$

Using our total correctness logic, and Theorem 8, it suffices to show, for any arbitrary positive ε, and any initial position n,

$$\vdash \langle \not{\varepsilon}(\varepsilon)\rangle\ \mathsf{spline}\ n\ \langle \mathsf{True}\rangle \tag{6}$$

The crucial part of the proof is the following auxiliary result, proven by induction on k

$$\vdash \left\langle \not{\varepsilon}\left(\frac{n}{n+k+1}\right)\right\rangle \mathsf{spline}\ n\ \langle \mathsf{True}\rangle$$

for all n. In the case $k = 0$ we own $\not{\varepsilon}\left(\frac{n}{n+1}\right)$, which is precisely what we need in position n to ensure we choose to stop. For the successor case, we own $\not{\varepsilon}\left(\frac{n}{n+(k+1)+1}\right)$. Here we can apply HT-RAND-EXP to ensure we either jump to 0 or we jump to $n + 1$ and scale our credits by $\frac{n+1}{n}$, which leaves us with $\lightning\left(\frac{n+1}{n+1+k+1}\right)$. This is what we require to apply our inductive hypothesis at position $n+1$, and finish the proof of this lemma.

Going back to proving Equation (6), since our initial budget $\lightning(\varepsilon)$ is strictly positive, and the sequence $\frac{n}{n+k+1}$ over k gets arbitrarily small, we can always pick a K such that $\frac{n}{n+K+1} < \varepsilon$, and then weaken our assumption to $\lightning\left(\frac{n}{n+K+1}\right)$ and conclude by applying the auxiliary lemma above.

To contrast with our proof, McIver et al. introduce a specialized rule to prove that this and other complex examples terminate almost surely. Within our setting, error credits and (meta-logic) induction can be used directly, with no need for additional rules. It is not clear if their other case studies can also be proven AST using error credits, or if their termination rule can be encoded in terms of error credits.

5.2.6 Additional Case Studies. While the planner rule is a versatile technique for proving almost-sure termination, it is not the only way to abstract *induction by error amplification*.

In particular, when the "target sample" has complex dependencies on program state it may be cumbersome to explicitly produce a target sample function z. In the long version we outline an alternative approach for proving almost-sure termination, which leverages a higher-order specification to directly express a relationship between the behavior of a sampler, checker, and error credit values. We then apply this specification to show that *WalkSAT*, a randomized SAT solver whose behavior is highly dependent on state, almost surely recognizes satisfiable 3SAT formulas.

6 Semantic Model and Soundness

We now turn our attention to the semantic model of Eris, which we use to prove soundness of the proof rules for Eris and to prove the adequacy theorem presented in §3.

Following standard practice [Jung et al. 2018], we define Eris Hoare triples in terms of a *weakest precondition* predicate

$$\{P\}\ e\ \{Q\} \triangleq \Box(P \twoheadrightarrow \mathsf{wp}\ e\ \{Q\})$$

where the persistence modality $\Box$ ensures that the predicate can be duplicated. However, our definition of the weakest precondition predicate $\mathsf{wp}\ e\ \{Q\}$ is novel. The definition is shown below. We omit from the definition the parts pertaining to how the Iris logic handles modifications to resources via "update modalities", since these details would distract from the definition and are completely standard. The full definition can be found in Appendix A.1.

$$\begin{aligned}
\mathsf{wp}\ e_1\ \{\Phi\} \triangleq\ & (e_1 \in Val \wedge \Phi(e_1)) \\
& \vee (e_1 \notin Val \wedge \forall \sigma_1, \varepsilon_1.\ S(\sigma_1) * \lightning_\bullet(\varepsilon_1) \twoheadrightarrow \\
& \qquad \mathsf{GLM}(e_1, \sigma_1, \varepsilon_1,\ (\lambda\, e_2, \sigma_2, \varepsilon_2\ .\ \triangleright(S(\sigma_2) * \lightning_\bullet(\varepsilon_2) * \mathsf{wp}\ e_2\ \{\Phi\}))))
\end{aligned}$$

The overall structure of this definition is similar to the weakest precondition for a non-probabilistic language [Jung et al. 2018, §6.3]. In particular, $\mathsf{wp}\ e_1\ \{\Phi\}$ is defined by guarded recursion. The first clause of the disjunction indicates that the weakest precondition for a value simply means that the postcondition $\Phi(e_1)$ must be satisfied. The second clause of the disjunction deals with the non-value case. It requires that the *state interpretation* $S(\sigma)$ is valid, which connects the logical points-to connectives to the physical state of the program. Both the heap and the presampling tapes are handled in this way, using the standard interpretation of state as partial finite maps from locations (resp. labels) to values (resp. presampled values) [Gregersen et al. 2024; Jung et al. 2018].

The weakest precondition gives meaning to ownership of the error resource $\lightning(\varepsilon)$ through the *error interpretation* $\lightning_\bullet(\varepsilon_1)$. Just like for the state interpretation, the error interpretation $\lightning_\bullet(\varepsilon_1)$ connects the logical connective for error credits $\lightning(\varepsilon)$ to the errors during program execution. Error credits are defined using the authoritative resource algebra [Jung et al. 2018, 2015b] over the positive real numbers with addition and the natural order, whose valid elements are the numbers in the half-open interval $[0, 1)$. The definition of the error credit resource is thus similar to that of later credits [Spies et al. 2022], but instead of $\mathsf{Auth}(\mathbb{N}, +)$ we use $\mathsf{Auth}(\mathbb{R}_0^+, +)$ with validity restricted to elements strictly smaller than 1. The proposition $\lightning(\varepsilon)$ asserts ownership of a fragmental element of the resource algebra, while $\lightning_\bullet(\varepsilon)$ stands for the authoritative view. The error rules from §3 then follow directly from the definition of the error credit resource together with the rules for the authoritative resource algebra.

The novel part of our definition of weakest precondition (besides the addition of the error interpretation) is that the recursive appearance of the weakest precondition is wrapped in the *graded lifting modality* GLM. The exact way in which GLM connects the operational semantics to errors will be explained in the next section. For now, we focus on its intuitive use in the weakest precondition. Think of (e_1, σ_1) as the starting configuration and of ε_1 as the current error budget. Through GLM, we quantify over the configurations we may step to according to the operational semantics as (e_2, σ_2), and ε_2 stands for the left-over error budget. The final part of the definition then indicates that the state and error interpretations with respect to σ_2 and ε_2 have to be satisfied, and that the weakest precondition has to hold recursively for ε_2 and Φ. Crucially, this recursive appeal to the weakest precondition occurs under the later modality $\triangleright$. This is what allows us to take the guarded fixed point of wp (which, in turn, allows us to prove soundness of the recursion rule HT-REC).

For Eris$_t$, the only difference is that we omit the $\triangleright$ modality in the definition of weakest precondition and instead define the predicate by the least fixed point (this is well-defined since wp only occurs positively inside its own definition).

6.1 The Graded Lifting Modality

We now turn our attention to the graded lifting modality $\mathsf{GLM}(e_1, \sigma_1, \varepsilon, Z)$. Eris uses the graded lifting modality to construct approximate predicate liftings of the graded predicate on configurations Z with respect to the distributions induced by the execution of (e_1, σ_1). As we shall see, to prove the modality, the initial error budget ε may be shared between the modality and Z. Our use of the graded lifting modality in the weakest precondition bears similarity with the coupling modality of Clutch [Gregersen et al. 2024], which was used to construct couplings between the execution of a specification program and its refinement, but the definition of our modality itself is rather different.

To focus the discussion on the most interesting aspects of the modality, we first present a simplified version STEP-SIMPLE that only supports reasoning about uniform error bounds. We then show how to modify the definition to enable expected error bound reasoning in STEP-EXP. The full definition, which additionally supports expected error reasoning for presampling tapes, can be found in Appendix A.2.

The simplified version is specified by the following rule (which should be read as a definition, expressing that the $\mathsf{GLM}(e_1, \sigma_1, \varepsilon, Z)$ predicate in the conclusion holds if the separating conjunction of the premises above the line holds):

$$\frac{\mathsf{red}(e_1, \sigma_1) \quad \varepsilon_1 + \varepsilon_2 \leq \varepsilon \quad \Pr_{\mathsf{step}(e_1,\sigma_1)}[\neg R] \leq \varepsilon_1 \quad \forall e_2, \sigma_2.\, R(e_2, \sigma_2) \twoheadrightarrow Z(e_2, \sigma_2, \varepsilon_2)}{\mathsf{GLM}(e_1, \sigma_1, \varepsilon, Z)}\ \textsc{step-simple}$$

The intuitive meaning of STEP-SIMPLE is that we can split the starting error budget ε into $\varepsilon_1 + \varepsilon_2$ and then likewise split the reasoning about the behaviour of the program into reasoning about the first step and about the rest of the execution separately. The error bounds can then be composed to yield a bound on the execution of the whole program.

On a technical level, the first premise of STEP-SIMPLE ensures that the program does not get stuck (red is short for reducible). The second premise states that the error budget can be split into two parts ε_1 and ε_2 provided their sum does not exceed ε. The inequality gives some flexibility in error accounting by allowing one to "weaken" the error bound: it is always sound to leave error budget unused. The user of the rule then picks an auxiliary intermediate predicate on configurations R. The premise $\mathrm{Pr}_{\mathrm{step}(e_1,\sigma_1)}[\neg R] \leq \varepsilon_1$ states that the configurations (e_2, σ_2) which (e_1, σ_1) can reduce to in one step do not violate R with error more than ε_1, *i.e.*, $\mathrm{Pr}_{\mathrm{step}(e_1,\sigma_1)}[\neg R] \leq \varepsilon_1$. Finally, the last premise requires a proof of Z for configurations (e_2, σ_2) with error budget ε_2, but in that proof we may now assume that $R(e_2, \sigma_2)$ is satisfied, since we "paid" for this assumption with ε_1.

Error in expectation. The rule STEP-SIMPLE imposes a constant bound on the error credit ε_2 that is left available for the correctness proof of the remainder of the program (e_2, σ_2). However, as we saw in the examples on expected error analysis, some expressions e_2 may need more or less error credit than others. This intuition is realized via the next rule.

$$\frac{\begin{array}{c}\mathrm{red}(\rho_1) \qquad \mathrm{Pr}_{\mathrm{step}(\rho_1)}[\neg R] \leq \varepsilon_1 \qquad \exists r.\, \forall \rho_2.\, \mathcal{E}_2(\rho_2) \leq r \\ \varepsilon_1 + \sum_{\rho_2 \in Cfg} \mathrm{step}(\rho_1)(\rho_2) \cdot \mathcal{E}_2(\rho_2) \leq \varepsilon \\ \forall \rho_2.\, R(e_2, \sigma_2) \mathrel{-\!\!*} \mathcal{E}_2(\rho_2) \geq 1 \vee Z(\rho_2, (\mathcal{E}_2(\rho_2)))\end{array}}{\mathsf{GLM}(\rho_1, \varepsilon, Z)}\ \textsc{step-exp}$$

The first two premises serve the same purpose as in STEP-SIMPLE, and the third premise is a purely technical side-condition that guarantees that the sum in premise four exists. The novelty in STEP-EXP is that instead of a fixed error for the "rest of the program", we have a configuration-indexed family of errors $\mathcal{E}_2$. Premise four states that the error budget ε can be split into ε_1 and, for each ρ_2 the starting configuration ρ_1 can step to, $\mathcal{E}_2(\rho_2)$ error credits, so long as the weighted sum of the errors multiplied by the probability of attaining each ρ_2 is below ε. This weighted sum is, of course, nothing other than the expectation of the random variable $\mathcal{E}_2$ over the distribution $\mathrm{step}(\rho_1)$. The last premise is similar to that of STEP-SIMPLE, except that STEP-EXP of course uses the rescaled error $\mathcal{E}_2(\rho_2)$. Another detail that was omitted from the simple rule is that we include a clause that allows us to conclude immediately if the remaining error budget exceeds 1.

6.2 Soundness, Adequacy, and Almost-Sure Termination

Using our model of weakest preconditions and Hoare triples, we can prove soundness of the program logic proof rules. For reasons of space, we refer the reader to the accompanying Coq formalization for details. The adequacy theorem for Hoare triples follows directly from the corresponding theorem for the weakest precondition:

THEOREM 9 (LIMIT WP ADEQUACY). *If* $\lightning(\varepsilon) \vdash \mathsf{wp}\ e\ \{\phi\}$ *then* $\forall \sigma.\ \mathrm{Pr}_{\mathrm{exec}(e,\sigma)}[\neg\phi] \leq \varepsilon$

Since exec is continuous in the sense that $(\forall n.\ \mathrm{Pr}_{\mathrm{exec}_n(e,\sigma)}[\phi] \leq x) \implies \mathrm{Pr}_{\mathrm{exec}\,(e,\sigma)}[\phi] \leq x$, it suffices to prove the corresponding statement about finite executions of arbitrary length. By applying the standard soundness theorem of the Iris base logic, we can thus restrict our attention to showing that $\vdash \triangleright^n\ \mathrm{Pr}_{\mathrm{exec}_n(e,\sigma)}[\neg\phi] \leq \varepsilon$ holds. The proof then proceeds by induction on the step index n. The inductive step for this argument hinges on the following lemma:

$$\mathsf{GLM}(\rho_1, \varepsilon_1, (\lambda(\rho_2, \varepsilon_2),\ \triangleright^{n+1}(\mathrm{Pr}_{\mathrm{exec}_n(\rho_2)}[\neg\phi] \leq \varepsilon_2))) \vdash \triangleright^{n+1}\ (\mathrm{Pr}_{\mathrm{exec}_{n+1}(\rho_1)}[\neg\phi] \leq \varepsilon_1)$$

Intuitively, this says that the graded lifting modality can be composed with an error bound for an n-step execution of the program ρ_1 to obtain an error bound on the execution of ρ_1 for $(n+1)$ steps. This should come as no surprise, since GLM requires the existence of an error bound on a single execution step. The key lemma that allows this composition is then the corresponding lemma for composing error bounds along monadic composition:

LEMMA 10. *Let* $\mu \in \mathcal{D}(A)$, *and let* f *be an* A*-indexed family of distributions, and let* $\mathcal{E}_2$ *be a family of errors. If* $\mathrm{Pr}_\mu[\neg\phi] \leq \varepsilon$ *and* $\exists x.\, \forall a.\ 0 \leq \mathcal{E}_2(a) \leq x$ *and* $(\forall a.\ \phi(a) \implies \mathrm{Pr}_{f(a)}[\neg\psi] \leq \mathcal{E}_2(a))$, *then* $\mathrm{Pr}_{\mu \gg\!= f}[\neg\psi] \leq \varepsilon_1 + \sum_{a \in A} \mu(a) \cdot \mathcal{E}_2(a)$.

This lemma in turn is proven by carefully re-arranging the terms of the sums obtained from the definition of the bind of the probability monad.

Finally, the almost-sure termination theorem for Eris_t (Theorem 8) is proved by (1) proving the total adequacy theorem in much the same manner as the partial adequacy theorem (except that no later modalities are involved) and (2) by the completeness of the real numbers, in the sense that for any $x, y \in \mathbb{R}$, if $\forall \varepsilon > 0.\, x - \varepsilon \leq y$ then $x \leq y$.

7 Related Work

Accuracy of probabilistic programs. Our logic is inspired by aHL [Barthe et al. 2016b], which introduced the idea of using a grading on Hoare triples that indicates the probability of the program failing to satisfy the specification, and then adding those errors through the sequence rule. This work considered an imperative probabilistic While language and used their approach to reason about accuracy of differentially private mechanisms. These ideas were then extended to the higher-order setting first by Sato et al. [2019], who consider a probabilistic lambda calculus with terminating recursion, and then by Aguirre et al. [2021], who add global first-order state via a state monad. Compared to them, we consider full recursion and higher-order state with dynamic allocation, and we validate new proof principles, including expected error composition and value dependent error.

Expectation preserving composition of error can be related to expectation-based logics, such as Batz et al. [2019]; Kaminski et al. [2016]; Morgan et al. [1996], where predicates are real-valued random variables. These logics are presented via weakest-precondition-style predicate transformers, and the weakest precondition of a sampling statement is precisely the expected value of its postcondition, similar to how credits are transformed in our HT-RAND-EXP rule. These logics can also be used to reason about approximate correctness, but they target first-order imperative languages. Recently, these techniques were applied in Batz et al. [2023] to reason about amortized expected time complexity of probabilistic programs. Various weakest pre-expectation-based logics also support techniques for proving almost-sure termination [Kaminski 2019, Chapter 6]. In a variant of one of these logics, McIver et al. [2018] present a powerful rule for proving almost-sure termination of probabilistic programs that are out of scope of other techniques. While it is unclear if it is possible to encode this general principle using error credits, we have used Eris_t to prove that their Example 5.4 is AST. The recently presented Caesar [Schröer et al. 2023] provides SMT-based support for verification in expectation-based logics.

Other approaches have tried to automate the computation of the probability that a program fails to satisfy a postcondition [Chakarov and Sankaranarayanan 2013; Smith et al. 2019; Wang et al. 2021]. These exploit different techniques of probability theory and programming language theory, such as martingales, concentration inequalities, approximants of fixed points, etc.

Approximate reasoning for probabilistic programs is also useful in the relational setting. Barthe et al. [2016a] introduce approximate couplings, which can be applied to prove different notions of approximate equivalence or differential privacy, in the setting of first-order imperative programs. Aguirre et al. [2021] also show that these techniques can be extended to the higher-order setting with global state. Also in the relational setting, the line of work on Rely [Carbin et al. 2013] considers two kinds of approximate properties: the probability that a program executes correctly, and the accuracy of the result itself.

Credit-based reasoning and resource analysis. There is a long line of work on automated amortized resource analysis [Hoffmann and Jost 2022; Hofmann and Jost 2003], which uses a substructural type system that associates a *potential* (a kind of stored credit) with a data structure. Recent work has extended this approach to probabilistic programs [Das et al. 2023; Ngo et al. 2018;

Wang et al. 2020] to prove bounds on expected costs. Analogously to our expected error rules, their typing rules for sampling instructions allow to average the potential across all possible outcomes.

Atkey [2011] proposes a realization of ARAA-style potentials as a separation logic resource, via a notion of *credit*. This idea was adapted to λ_{ref} and implemented in Coq by Charguéraud and Pottier [2019], and later brought to Iris [Mével et al. 2019; Pottier et al. 2024]. Error credits follow a parallel story, namely realizing aHL-style error annotations on triples as a separation logic resource and exploring the gains on expressive power we obtain. However, there are no further similarities between the implementations of error credits and time credits. In particular, the semantics of the languages, the property being tracked and the approach to proving soundness are all different.

Probabilities and separation logic. A number of works in recent years have focused on the interactions between separation logic and probabilities. Gregersen et al. [2024] introduced Clutch, upon which we build. They present a separation logic to reason about higher-order probabilistic programs, focusing on relational properties and in particular contextual equivalence. Batz et al. [2019] present an expectation-based version of separation logic, which can be used to prove error bounds for first-order pointer programs.

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.

Other works focus on reinterpreting the notion of separating conjunction in separation logic to represent probabilistic independence. This line of work originated with Barthe et al. [2020], and different variants have been developed (Bao et al. [2022]; Li et al. [2023]). These works also focus on first-order programs. In our work, the separating conjunction has the standard meaning, it is only Hoare triples as whole that have a probabilistic interpretation. Exploring a deeper connection between our logic and separation logics for independence would be an interesting follow-up.

8 Conclusions and Future Work

In this paper we presented Eris, which develops the idea of representing error as a resource to enable novel reasoning principles for approximation bounds that lead to more modular and precise specifications compared to prior work, including almost sure termination of probabilistic algorithms.

There are multiple directions for future work. Firstly, it would be interesting to extend Eris to a concurrent language, to support reasoning about approximate randomized concurrent algorithms. Secondly, the idea of expected error composition should apply to other kinds of separation logic resources, such as time credits, and could be used to reason about expected time complexity of higher-order probabilistic programs. Thirdly, by integrating ideas from separation logics for probabilistic independence, we could encode concentration bounds that exploit this independence and thereby obtain more precise error bounds. Finally, we believe our ideas should also apply to the relational setting, where the error credits could be used to prove approximate couplings, and have interesting applications to security and differential privacy.

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.

A Appendix: Full Definition of the Weakest Precondition and Graded Lifting Modality

The definitions of the weakest precondition and of the graded lifting modality as presented in §6 contain some simplifications for the sake of pedagogy. We now restate the definitions in full detail.

A.1 The Weakest Precondition in Detail

The full definition of the weakest precondition differs from the one presented in §6 in that it also contains the invariant mask annotation and fancy update modality of Iris.

$$\begin{aligned}\mathrm{wp}_{\mathcal{E}}\; e_1\; \{\Phi\} \triangleq\; & (e_1 \in \mathit{Val} \wedge \Rrightarrow_{\mathcal{E}} \Phi(e_1)) \vee \\ & (e_1 \notin \mathit{Val} \wedge \forall \sigma_1, \varepsilon_1.\, S(\sigma_1) * \lightning_\bullet(\varepsilon_1) \mathrel{-\!\!*} {}_{\mathcal{E}}\Rrightarrow_{\emptyset} \\ & \quad \mathsf{GLM}\; (e_1, \sigma_1, \varepsilon_1,\; (\lambda\, e_2, \sigma_2, \varepsilon_2\,.\, \triangleright\, {}_{\emptyset}\Rrightarrow_{\mathcal{E}} (S(\sigma_2) * \lightning_\bullet(\varepsilon_2) * \mathrm{wp}_{\mathcal{E}}\; e_2\; \{\Phi\})))\end{aligned}$$

Just as before, the full definition of the total weakest precondition is obtained by omitting the later modality on the last line, and by taking the least instead of the guarded fixed point of the recursive definition.

A.2 The Graded Lifting Modality in Detail

In §6.1, we presented a simplified version of the graded lifting modality which does not support presampling tapes. The full definition of GLM contains two clauses: STEP-EXP for expected error lifting of program steps and STATESTEP-EXP for the presampling analog.

The rules in this section should be read as defining GLM as an inductive predicate, *i.e.* as the least fixed point of the closure system associated to the rules.

Adding presampling tapes. The presampling ghost operations on tapes are realized through an auxiliary *state steps* relation state_step : $\mathit{Label} \times \mathit{State} \to \mathcal{D}(\mathit{State})$. If ι is the label associated to an (allocated) tape with bound N, then $\mathrm{state\_step}_\iota(\sigma_1)$ denotes the distribution on states obtained by appending a uniformly randomly sampled value between 0 and N to the end of tape ι:

$$\mathrm{state\_step}_\iota(\sigma_1)(\sigma_2) = \begin{cases} \frac{1}{N+1} & \text{if } \sigma_2 = \sigma_1[\iota \to (N, \vec{n} \cdot n)] \text{ and } \sigma_1(\iota) = (N, \vec{n}) \text{ and } n \leq N, \\ 0 & \text{otherwise.} \end{cases}$$

We can now extend the graded lifting modality to allow taking state steps. Just as we did with STEP-SIMPLE, we will first discuss a simplified rule that does not support reasoning about errors in expectation. In practice, this rule is derivable from the rule STATESTEP-EXP below.

$$\frac{\varepsilon_1 + \varepsilon_2 \leq \varepsilon \qquad \mathrm{Pr}_{\mathrm{state\_step}_\iota(\sigma_1)}[\neg R] \leq \varepsilon_1 \qquad \forall \sigma_2.\, R(\sigma_2) \mathrel{-\!\!*} \mathsf{GLM}(e_1, \sigma_2, \varepsilon_2, Z)}{\mathsf{GLM}(e_1, \sigma_1, \varepsilon, Z)}\ \text{STATESTEP-SIMPLE}$$

The user of the rule can, once again, split the error budget ε between a first step and the remainder of the program. In STATESTEP-SIMPLE, however, the first step is a state step, *i.e.* a purely logical step in the ghost state. The recursive occurrence of GLM in the last premise of STATESTEP-SIMPLE allows the user of the modality to perform a number of state steps before eventually proving the base case of the modality, namely STEP-EXP.

Finally, we can combine the idea of reasoning of expected-error reasoning with state steps via the following rule.

$$\dfrac{\begin{array}{c}\mathrm{red}(e_1,\sigma_1) \qquad \Pr_{\mathrm{state\_step}_\iota(\sigma_1)}[\neg R] \le \varepsilon_1 \qquad \exists r.\, \forall \rho_2.\, \mathcal{E}_2(\rho_2) \le r \\ \varepsilon_1 + \displaystyle\sum_{\sigma_2 \in State} \mathrm{state\_step}(e_1,\sigma_1)(e_1,\sigma_2) \cdot \mathcal{E}_2(e_1,\sigma_2) \le \varepsilon \\ \forall \sigma_2.\, R(\sigma_2) \twoheadrightarrow \mathcal{E}_2(e_1,\sigma_2) \ge 1 \vee \mathrm{GLM}(e_1,\sigma_2,\mathcal{E}_2(e_1,\sigma_2))\end{array}}{\mathrm{GLM}(e_1,\sigma_1,\varepsilon,Z)}\ \textsc{statestep-exp}$$

This rule is used, for instance, to derive the program logic rule PRESAMPLE-EXP.

B Appendix: Additional Details for Collision-Free Hash Map

We include the code for the insertion function for a collision-free, resizing hash map, as studied in §4.4. A hash map is represented as a tuple (l, hf, v, s, r), where l is an array containing the physical representation of the map, hf is a (collision-free) hash function as shown in §4.3, v is the current size of the array containing the hash map, s is the number of initialized entries and r is the size threshold before resizing:

$$\begin{array}{rl} \mathit{insert}\ \mathit{hm}\ w \triangleq & \mathsf{let}\ (l, hf, v, s, r) = hm\ \mathsf{in} \\ & \mathsf{let}\ (b, hf') = \mathsf{hash\_rs}\ hf\ w\ \ \mathsf{in} \\ & \mathsf{let}\ w' = !\,l[b]\ \ \mathsf{in} \\ & \mathsf{if}\ w' = ()\ \mathsf{then} \\ & \quad l[b] \leftarrow w \\ & \quad \mathsf{if}\ s+1 = r\ \mathsf{then} \\ & \quad\quad \mathsf{let}\ l' = \mathit{resize}\ l\ v\ v\ \ \mathsf{in} \\ & \quad\quad (l', hf', 2 * v, s+1, 2 * r) \\ & \quad \mathsf{else}\ (l, hf', v, s+1, r) \\ & \mathsf{else}\ (l, hf', v, s, r) \end{array}$$

Note in particular that if we try to insert an element w and there is another element w' with the same hash, then w will not get inserted into the table. However, the specification of hash_rs ensures that this will not happen if we have ownership of $\lightning((3 \cdot R_0)/(4 \cdot V_0))$.

The core part of the representation predicate for the hash map is shown below:

$$\begin{array}{rl} \mathrm{isHashmap}\ hm\ ns \triangleq & \exists l, hf, v, s, r, m, tbl. (hm = (l, hf, v, s, r)) * \\ & l \mapsto^* tbl * ((\mathit{filterUnits}\ tbl) \equiv ns) * \\ & \mathit{cf\_hash\_rs}\ hf\ \ m\ v\ s\ r * \\ & (\forall (i, w : \mathbb{N}). m[w] = i \leftrightarrow tbl[i] = w) * \\ & (\forall i < v, i \notin \mathrm{img}\ m \rightarrow tbl[i] = ()) * (\dots) \end{array}$$

This should be read as "*hm* is a hash map representing the set (of natural numbers) *ns*". The hash map contains a table *tbl* whose contents are either natural numbers or unit, and the set of natural numbers it contains is exactly *ns*. The index at which every element is located is controlled by a collision-free hash function hf, that tracks a partial map m. Thus the table will contain an element w at index i if and only if m maps w to i.

C Appendix: Expectation-Preserving Composition on Words

While HT-RAND-EXP and PRESAMPLE-EXP can move error credits between the outcomes of a single random event, in order to prove the planner rule we need to move error credits out of the sequence of events which sample an entire target word. Defining a suitable sequence of expectation-preserving composition steps to accomplish this can be subtle: sampling any prefix of our target word should *decrease* our error credit, but sampling a prefix of the target followed by an erroneous sample should yield an amplification on our *initial* credit amount.

Let $\vec{w}$ be a word of length $L > 0$ in the alphabet $[0, N]$. For $0 \le i \le L$, define the constants

$$\mathit{ecAmp}_{N,L} \triangleq 1 + \frac{1}{(N+1)^L - 1} \qquad \mathit{ecRem}_{N,L}(i) \triangleq 1 - \frac{(N+1)^i - 1}{(N+1)^L - 1}$$

so that $0 \le \mathit{ecRem}_{N,L}(i) < 1 < \mathit{ecAmp}_{N,L}$. Suppose we want to amplify some positive amount of credit ε against $\vec{w}$; that is we seek to either sample all of $\vec{w}$, or obtain extra error credits. For $0 \le i < L$, define the error distribution functions

$$D^{\varepsilon}_{N,L}(i, c) \triangleq \begin{cases} \mathit{ecRem}_{N,L}(i+1) \cdot \varepsilon & c = \vec{w}[i] \\ \mathit{ecAmp}_{N,L} \cdot \varepsilon & \text{otherwise} \end{cases}$$

Starting with $↯\left(\mathit{ecRem}_{N,L}(i) \cdot \varepsilon\right)$ the function $D^{\varepsilon}_{N,L}(i, \_)$ is mean-preserving, since

$$\begin{aligned} \sum_{c=0}^{N} D^{\varepsilon}_{N,L}(i, c) &= \mathit{ecRem}_{N,L}(i+1) \cdot \epsilon + N \cdot \mathit{ecAmp}_{N,L} \cdot \varepsilon \\ &= \left(\frac{(N+1)(N+1)^L - (N+1)^{i+1}}{(N+1)^L - 1} \right) \cdot \varepsilon \\ &= (N+1) \cdot \mathit{ecRem}_{N,L}(i) \cdot \varepsilon \end{aligned}$$

Now we can redistribute the error credit out of the event where we sample $\vec{w}$ and distribute it evenly into all other cases, using $L - 1$ steps of advanced composition. Starting with $i = 0$, at the beginning of the i^{th} sample we have will have correctly sampled the first i characters of $\vec{w}$ and own $↯\left(\mathit{ecRem}_{N,L}(i) \cdot \varepsilon\right)$. At step i, perform expectation preserving composition using the error function $D^{\varepsilon}_{N,L}(i, \_)$. Each composition either correctly samples the next character of $\vec{w}$ and decreases the error credit supply to $↯\left(\mathit{ecRem}_{N,L}(i+1) \cdot \varepsilon\right)$, or increases it to $↯\left(\mathit{ecAmp}_{N,L} \cdot \varepsilon\right)$. Note that $↯\left(\mathit{ecRem}_{N,L}(0) \cdot \varepsilon\right) = ↯(\varepsilon)$ for the initial case, and $↯\left(\mathit{ecRem}_{N,L}(L) \cdot \varepsilon\right) = ↯(0)$ once $\vec{w}$ is completely sampled. In aggregate, this sequence of proof steps will either result in sampling $\vec{w}$ or increasing our error credit by a factor of $\mathit{ecAmp}_{N,L}$.

Implemented using PRESAMPLE-EXP, this procedure proves the amplification lemma from §5.2.4:

$$\frac{\left\langle \exists \vec{j}.\, \iota \hookrightarrow (N, \vec{n} +\!\!+ \vec{j}) * ↯\left(\mathit{ecAmp}_{N,L} \cdot \varepsilon\right) \right\rangle e \left\langle \phi \right\rangle \qquad \left\langle \iota \hookrightarrow (N, \vec{n} +\!\!+ \vec{w}) \right\rangle e \left\langle \phi \right\rangle}{\left\langle \iota \hookrightarrow (N, \vec{n}) * ↯(\varepsilon) \right\rangle e \left\langle \phi \right\rangle}$$

Finally, it will be convenient to define a lower bound on the amount of extra credit generated each time our chain of advanced composition fails to sample $\vec{w}$: $\mathit{ecExc}_{N,L} \triangleq \mathit{ecAmp}_{N,L} - 1$. Since $\mathit{ecRem}_{N,L}(i) < 1$ for all $0 \le i \le L$, we can prove that

$$↯\left(\mathit{ecAmp}_{N,L}\right) \twoheadrightarrow ↯\left(\mathit{ecRem}_{N,L}(i)\right) * ↯\left(\mathit{ecExc}_{N,L}\right) \tag{7}$$

In other words, when we fail to sample $\vec{w}$ using this technique we have at least enough credit to try again, plus an additional $↯\left(\mathit{ecExc}_{N,L} \cdot \varepsilon\right)$.

References

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

Alejandro Aguirre, Philipp G. Haselwarter, Markus de Medeiros, Kwing Hei Li, Simon Oddershede Gregersen, Joseph Tassarotti, and Lars Birkedal. 2024. *Error Credits: Resourceful Reasoning about Error Bounds for Higher-Order Probabilistic Programs - Coq Artifact.* https://doi.org/10.5281/zenodo.11489778

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

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

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

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

Gilles Barthe, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2016b. A Program Logic for Union Bounds. In *43rd International Colloquium on Automata, Languages, and Programming (ICALP 2016).* Schloss-Dagstuhl - Leibniz Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.ICALP.2016.107

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

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

Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Lena Verscht. 2023. A Calculus for Amortized Expected Runtimes. *Proc. ACM Program. Lang.* 7, POPL (2023), 1957–1986. https://doi.org/10.1145/3571260

Mihir Bellare and Phillip Rogaway. 1993. Random Oracles are Practical: A Paradigm for Designing Efficient Protocols. In *CCS '93, Proceedings of the 1st ACM Conference on Computer and Communications Security, Fairfax, Virginia, USA, November 3-5, 1993*, Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, Ravi S. Sandhu, and Victoria Ashby (Eds.). ACM, 62–73. https://doi.org/10.1145/168588.168596

Juan Benet. 2014. IPFS - Content Addressed, Versioned, P2P File System. *CoRR* abs/1407.3561 (2014). arXiv:1407.3561 http://arxiv.org/abs/1407.3561

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

Michael Carbin, Sasa Misailovic, and Martin C. Rinard. 2013. Verifying quantitative reliability for programs that execute on unreliable hardware. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications* (Indianapolis, Indiana, USA) *(OOPSLA '13).* Association for Computing Machinery, New York, NY, USA, 33–52. https://doi.org/10.1145/2509136.2509546

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 (Lecture Notes in Computer Science, Vol. 8044)*, Natasha Sharygina and Helmut Veith (Eds.). Springer, 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. *Journal of Automated Reasoning* 62, 3 (March 2019), 331–365. https://doi.org/10.1007/s10817-017-9431-7

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, 3rd Edition.* MIT Press. http://mitpress.mit.edu/books/introduction-algorithms

Ankush Das, Di Wang, and Jan Hoffmann. 2023. Probabilistic Resource-Aware Session Types. *Proc. ACM Program. Lang.* 7, POPL (2023), 1925–1956. https://doi.org/10.1145/3571259

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, Thomas C. Bressoud and M. Frans Kaashoek (Eds.). ACM, 205–220.

https://doi.org/10.1145/1294261.1294281

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

Jan Hoffmann and Steffen Jost. 2022. Two decades of automatic amortized resource analysis. *Math. Struct. Comput. Sci.* 32, 6 (2022), 729–759. https://doi.org/10.1017/S0960129521000487

Martin Hofmann and Steffen Jost. 2003. Static prediction of heap space usage for first-order functional programs. In *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, January 15-17, 2003*, Alex Aiken and Greg Morrisett (Eds.). ACM, 185–197. https://doi.org/10.1145/604131.604148

Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. *SIGPLAN Not.* 51, 9 (sep 2016), 256–269. https://doi.org/10.1145/3022670.2951943

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

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015a. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *ACM-SIGACT Symposium on Principles of Programming Languages*. https://api.semanticscholar.org/CorpusID:1174404

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015b. 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. 2019. *Advanced weakest precondition calculi for probabilistic programs*. Ph. D. Dissertation. RWTH Aachen University, Germany. http://publications.rwth-aachen.de/record/755408

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 (Lecture Notes in Computer Science, Vol. 9632)*, Peter Thiemann (Ed.). Springer, 364–389. https://doi.org/10.1007/978-3-662-49498-1_15

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

John M. Li, Amal Ahmed, and Steven Holtzen. 2023. Lilac: A Modal Separation Logic for Conditional Probability. *Proc. ACM Program. Lang.* 7, PLDI (2023), 148–171. https://doi.org/10.1145/3591226

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

Ralph C. Merkle. 1987. A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987, Proceedings (Lecture Notes in Computer Science, Vol. 293)*, Carl Pomerance (Ed.). Springer, 369–378. https://doi.org/10.1007/3-540-48184-2_32

Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2019. Time Credits and Time Receipts in Iris. In *Programming Languages and Systems (Lecture Notes in Computer Science)*, Luís Caires (Ed.). Springer International Publishing, Cham, 3–29. https://doi.org/10.1007/978-3-030-17184-1_1

Gary L. Miller. 1975. Riemann's Hypothesis and tests for primality. In *Proceedings of the Seventh Annual ACM Symposium on Theory of Computing* (Albuquerque, New Mexico, USA) *(STOC '75)*. Association for Computing Machinery, New York, NY, USA, 234–239. https://doi.org/10.1145/800116.803773

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

Van Chan Ngo, Quentin Carbonneaux, and Jan Hoffmann. 2018. Bounded expectations: resource analysis for probabilistic programs. 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

Michael O Rabin. 1980. Probabilistic algorithm for testing primality. *Journal of Number Theory* 12, 1 (Feb. 1980), 128–138. https://doi.org/10.1016/0022-314x(80)90084-0

Tetsuya Sato, Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Justin Hsu. 2019. Formal Verification of Higher-Order Probabilistic Programs: Reasoning about Approximation, Convergence, Bayesian Inference, and Optimization. *Proc. ACM Program. Lang.* 3, POPL (2019), 38:1–38:30. https://doi.org/10.1145/3290351

Philipp Schröer, Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. 2023. A Deductive Verification Infrastructure for Probabilistic Programs. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 2052–2082. https://doi.org/10.1145/3622870

Calvin Smith, Justin Hsu, and Aws Albarghouthi. 2019. Trace abstraction modulo probability. *Proc. ACM Program. Lang.* 3, POPL (2019), 39:1–39:31. https://doi.org/10.1145/3290352

R. Solovay and V. Strassen. 1977. A Fast Monte-Carlo Test for Primality. *SIAM J. Comput.* 6, 1 (1977), 84–85. https://doi.org/10.1137/0206006

Simon Spies, Lennard Gäher, Joseph Tassarotti, Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2022. Later credits: resourceful reasoning for the later modality. *Proc. ACM Program. Lang.* 6, ICFP (2022), 283–311. https://doi.org/10.1145/3547631

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

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

Jinyi Wang, Yican Sun, Hongfei Fu, Krishnendu Chatterjee, and Amir Kafshdar Goharshady. 2021. Quantitative Analysis of Assertion Violations in Probabilistic Programs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 1171–1186. https://doi.org/10.1145/3453483.3454102

Received 2024-02-28; accepted 2024-06-18