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 are thus only approximately correct. It is important to obtain precise error bounds for these approximations, but existing approaches rely on simplifications that make the error bounds excesively coarse, or only apply to first-order programs. In this paper we present Eris, a higher-order separation logic 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 the error bound of a program. 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 proving almost-sure termination by induction on the error.

We illustrate the advantages of our approach by proving amortized error bounds on a range of examples, including collision probabilities in hash functions, which allows 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.

1 INTRODUCTION

Approximate behaviour is ubiquitous in probabilistic programs. For example, a Monte Carlo algorithm trades *accuracy* for *efficiency*: we accept a small probability of obtaining a wrong result if it allows us to efficiently compute a correct result most of the time. Dually, a Las Vegas algorithm returns only correct results, but may, again with small probability, fail to find a result at all.

Both of these techniques are used pervasively in application domains of probabilistic programs, be it cryptography, machine learning, algorithms and data structures, statistical modeling or others. 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 with either "probably prime" or with "certainly composite". Similarly, 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).

Since the trade-off between efficiency and accuracy/termination is typically justified by the observation that the failure behaviour occurs with small probability, establishing error bounds is an important prerequisite for the use of Monte Carlo and Las Vegas algorithms. However, probabilistic reasoning in general 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 randomised programs. For randomised first-order WHILE programs, the approximate Hoare logic (aHL) of Barthe et al. [2016b]

---

provides a convenient way to over-approximate the error behaviour of an algorithm. Formally, aHL triples are annotated with an "error budget" ε; the judgement $\vdash_\varepsilon \{P\}\, c\, \{Q\}$ 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 formalised in the rule for sequential composition:

$$\frac{\vdash_{\varepsilon_1} \{P\}\, c_1\, \{Q\} \qquad \vdash_{\varepsilon_2} \{Q\}\, c_2\, \{R\}}{\vdash_{\varepsilon_1 + \varepsilon_2} \{P\}\, c_1; c_2\, \{R\}} \text{ AHL SEQ}$$

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 judgemental structure of the logic rather than treating them as ordinary propositions, these works on approximate correctness forego 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 error-annotated Hoare-triples are a judgement, and not a first class proposition which may itself occur in a precondition.

Furthermore, reasoning about composition via union bounds over-approximates error, and can produce excessively coarse bounds when errors are not independent. Specifically, the union 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 supporting advanced reasoning principles for proving specifications that hold up to error bounds, for programs written in $\lambda_{\text{ref}}^{\text{rand}}$, an expressive ML-like language with random sampling, full recursion, higher-order functions and higher-order store. Inspired by the use of time credits [Atkey 2011; Charguéraud and Pottier 2019; Mével et al. 2019] to reason about cost as a resource, we introduce and develop a resource point of view of error probabilities as *error credits*. Ownership of ε error credits is a first-class proposition in Eris, written as $\mathbf{\xi}(\varepsilon)$ (read: "up to ε"), and proving an Eris specification $\vdash \{P * \mathbf{\xi}(\varepsilon)\}\, e\, \{Q\}$ intuitively means: "if P holds then the result of evaluating e satisfies Q up to error ε".

This resource treatment affords Eris great flexibility: if we own an error credit $\mathbf{\xi}(\varepsilon)$, 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 $\mathbf{\xi}(\varepsilon_1), \ldots, \mathbf{\xi}(\varepsilon_n)$ 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 randomised 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 shown below for an iterator function List.iter. In the precondition, line (1) states that the argument l represents the mathematical sequence of values xs. Then, line (2) assumes a specification for the function e to be iterated over the list: for each argument x, e takes $R(x)$ as a precondition and returns $Q(x)$ as a postcondition up to error $\mathcal{E}(x)$. Notably, the error can depend on the value of x. Line (3) states that

---

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

for the elements of the list xs, the precondition R holds for each x in the list, and that the user of the List.iter should have enough error credits to "pay" for each call to e. Finally, the postcondition states that Q holds for each x in the list xs.

$$\left\{ \begin{array}{cc} (1) & \quad isList\ xs\ l\ * \\ (2) & \forall x.\ \{R(x) * \mathcal{f}(\mathcal{E}(x))\}\ e\ x\ \{Q(x)\}\ * \\ (3) & \quad \varhexstar_{x \in xs}\ (R(x) * \mathcal{f}(\mathcal{E}(x))) \end{array} \right\} \text{List.iter}\ e\ l \left\{ \varhexstar_{x \in xs}\ Q(x) \right\}$$

It is noteworthy that this specification is almost exactly the specification one would give for List.iter in a standard, non-probabilistic, higher-order separation logic, which looks as follows:

$$\left\{ \begin{array}{c} isList\ xs\ l\ * \\ \forall x.\ \{P(x)\}\ e\ x\ \{Q(x)\}\ * \\ \varhexstar_{x \in xs}\ P(x) \end{array} \right\} \text{List.iter}\ e\ l \left\{ \varhexstar_{x \in xs}\ Q(x) \right\}$$

Indeed, by treating error credits as a resource, the error credit version can be *derived* from the standard version by instantiating $P(x)$ to be $R(x) * \mathcal{f}(\mathcal{E}(x))$

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

$$\frac{\vdash \{P * \mathcal{f}(\varepsilon_1)\}\ e_1\ \{x.\ \mathcal{f}(\mathcal{E}_2(x)) * Q\} \qquad \vdash \forall x.\ \{Q * \mathcal{f}(\mathcal{E}_2(x))\}\ e_2\ \{R\}}{\vdash \{P * \mathcal{f}(\varepsilon_1)\}\ \text{let}\ x = e_1\ \text{in}\ e_2\ \{R\}} \text{ HT-BIND-EXP}$$

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. 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 is enough 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 as further discussed in §3.

Putting 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. Let e be the expression let $n = e_1$ in e_2, where $e_1 = \text{rand}\ K$, and $e_2(n) = \text{let}\ l = \text{List.make}\ 0\ n\ \text{in List.iter}\ e\ l$. Here we intend that List.make $0\ n$ constructs a list of zeros of length n, and that e is a function to be iterated, satisfying an assumption like (2) above. Using the HT-BIND-EXP rule, we can then prove a Hoare triple for e if we have $\varepsilon_1 = \varepsilon_e \cdot \frac{K}{2}$ credits in our precondition and set $\mathcal{E}_2(n) = n \cdot \varepsilon_e$. Note that ε_1 is the expected value of $\mathcal{E}_2$, and in combination, the two observations mean that we obtain a form of *expectation-preserving composition*.

Amortised 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 *amortised error bounds* for operations of randomised data structures. Atkey [2011] pioneered the use of "time credits"

to reason about amortized time complexity in separation logic, and the idea was subsequently extended and formalised in different separation logics [Charguéraud and Pottier 2019; Mével et al. 2019]. Our use of error credits in turn allows us to give modular, amortised specifications of randomised 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 amortised error bounds.

Almost-sure termination via error credits. So far, we have implicitly considered a partial correctness interpretation of the Eris Hoare triples. In particular this means that an always 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 via 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 via 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 turn, if we can show a total Hoare triple for *any* error bound ε, then we can conclude that it holds when ε becomes vanishingly small and thus that the program almost-surely terminates. 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 summarise, we provide:
- The first probabilistic higher-order separation logic, Eris, for *modular* approximate reasoning (up-to-errors) about probabilistic programs written in $\lambda_{\text{ref}}^{\text{rand}}$, a randomised 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, amortised error bounds.
- A total correctness version of Eris, which can be used to establish *lower* bounds on probabilities of program behaviours, 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 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 $\lambda_{\text{ref}}^{\text{rand}}$, and then we 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 appendix). Finally, 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 $\lambda_{\text{ref}}^{\text{rand}}$

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

2.1 Probabilities of Programs

As a first approximation, one might expect the execution of a randomised program e to produce a (discrete) probability distribution on values. However, since programs may not terminate, programs might not introduce *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* $f : X \to \mathbb{R}^{\geq 0}$ *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* $\mu : X \to [0, 1]$ *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 text. Unless otherwise specified, the variable μ denotes a distribution, and X a countable set, typically the set of values, expressions, or configurations of $\lambda_{\text{ref}}^{\text{rand}}$.

LEMMA 3 (PROBABILITY MONAD). *Let* $\mu \in \mathcal{D}(A)$, $a \in A$, *and* $f : A \to \mathcal{D}(B)$. *Then*

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

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

gives a monadic structure to $\mathcal{D}$. *We write* $\mu \ggg f$ *for* $\text{bind}(f, \mu)$.

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* $\text{Pr}_\mu[P]$, *is the total probability mass of* μ *satisfying* P, *i.e.* $\text{Pr}_\mu[P] = |\mu_{|P}|$.

The approximate correctness of programs is formulated in terms of graded predicate liftings defined below.

DEFINITION 6 (GRADED PREDICATE LIFTING). *Let* P *be a predicate and* μ *be a distribution, and* $\varepsilon \in [0, 1]$. *The* partial graded lifting *of* P *with respect to the grading* ε *is defined as* $\text{Pr}_\mu[\neg P] \leq \varepsilon$, *and written as* $\text{PGL}_\mu^\varepsilon[P]$. *The* total graded lifting *is given by* $\text{Pr}_\mu[P] \geq 1 - \varepsilon$, *and written as* $\text{TGL}_\mu^\varepsilon[P]$.

Note that, while the notions of partial and total graded predicate lifting coincide for proper distributions, this is not the case in general for subdistributions.

2.2 Language Definition and Operational Semantics

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

$$v, w \in Val ::= z \in \mathbb{Z} \mid b \in \mathbb{B} \mid () \mid \ell \in Loc \mid \text{rec f x} = e \mid (v, w) \mid \text{inl } v \mid \text{inr } v$$

$$e \in Expr ::= v \mid \text{x} \mid \text{rec f x} = e \mid e_1 \, e_2 \mid e_1 + e_2 \mid e_1 - e_2 \mid \dots \mid \text{if } e \text{ then } e_1 \text{ else } e_2 \mid$$

$$(e_1, e_2) \mid \text{fst } e \mid \text{snd } e \mid \text{inl}(e) \mid \text{inr}(e) \mid \text{match } e \text{ with inl } v \implies e_1 \mid \text{inr } w \implies e_2 \text{ end} \mid$$

$$\text{allocn } e_1 \, e_2 \mid \, ! \, e \mid e_1 \leftarrow e_2 \mid \text{rand } e$$

$$K \in Ectx ::= \; - \; \mid e \, K \mid K \, v \mid \text{allocn } K \mid \, ! \, K \mid e \leftarrow K \mid K \leftarrow v \mid \text{rand } K \mid \dots$$

$$\sigma \in State \triangleq (Loc \xrightarrow{\text{fin}} Val) \qquad \qquad \rho \in Cfg \triangleq Expr \times State$$

The term language is mostly standard: $\text{allocn } e_1 \, e_2$ 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 $\text{rec \_ } x = e$, let-bindings $\text{let } x = e_1 \text{ in } e_2$ defined as $(\lambda x. \, e_2) \, e_1$, sequencing $e_1; e_2$ defined as $\text{let \_} = e_1 \text{ in } e_2$, and references $\text{ref } e$ defined as $\text{allocn } e \; 1$. 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.[2] States in our language represent the heap as a finite map from memory locations to values.

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

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

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

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

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

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:

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

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

As an example, consider the program $e \triangleq \text{if flip \&\& flip then } 42 \text{ else } \Omega$, 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, $\text{exec } e$ induces the subdistribution $\{42 \mapsto 0.25, \_ \mapsto 0\} : Val \to [0, 1]$.

---

[2]As in Clutch, we support presampling tapes, but since they are not used until §5, we relegate their discussion to §5.2.2.

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 it means semantically to prove 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:

$$P, Q \in iProp ::= \mathsf{True} \mid \mathsf{False} \mid P \wedge Q \mid P \vee Q \mid P \Rightarrow Q \mid \forall x.\, P \mid \exists x.\, P \mid P * Q \mid P \multimap Q \mid$$

$$\Box P \mid \triangleright P \mid \boxed{P}^{\mathcal{N}} \mid \ell \mapsto v \mid \sharp(\varepsilon) \mid \{P\}\, e\, \{Q\} \mid \dots$$

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

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

From the point of view of a user of the logic, this interface 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 it is always 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, this would allow us to prove that the probability of some event is below 1, which is trivially true.

We define the semantics of our 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 7 (ADEQUACY). *If* $\vdash \{\sharp(\varepsilon)\}\, e\, \{\phi\}$ *then* $\mathrm{Pr}_{\mathrm{exec}\, e}[\neg\phi] \leq \varepsilon$

The adequacy theorem states that if we prove $\{\sharp(\varepsilon)\}\, e\, \{\phi\}$ in Eris, for any meta-logic postcondition ϕ, then the final distribution obtained from running e does *not* satisfy ϕ with at most probability ε.

As mentioned in the introduction, 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 get 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 $\lambda_{\mathrm{ref}}^{\mathrm{rand}}$ are essentially standard, and one can validate the same set of proof rules as for the sequential fragment of HeapLang, including, *e.g.*,

HT-FRAME
$$\frac{\vdash \{P\}\, e\, \{Q\}}{\vdash \{P * R\}\, e\, \{Q * R\}}$$

HT-BIND
$$\frac{\vdash \{P\}\, e\, \{v.Q\} \qquad \vdash \forall v.\, \{Q\}\, K[v]\, \{R\}}{\vdash \{P\}\, K[e]\, \{R\}}$$

HT-LOAD
$$\frac{\ell \mapsto w \vdash P(w)}{\vdash \{\ell \mapsto w\}\, !\ell\, \{v.\, P(v)\}}$$

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

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 the introduction, allows us to derive aHL-style composition rule as well as the the value-dependent composition rule shown in the
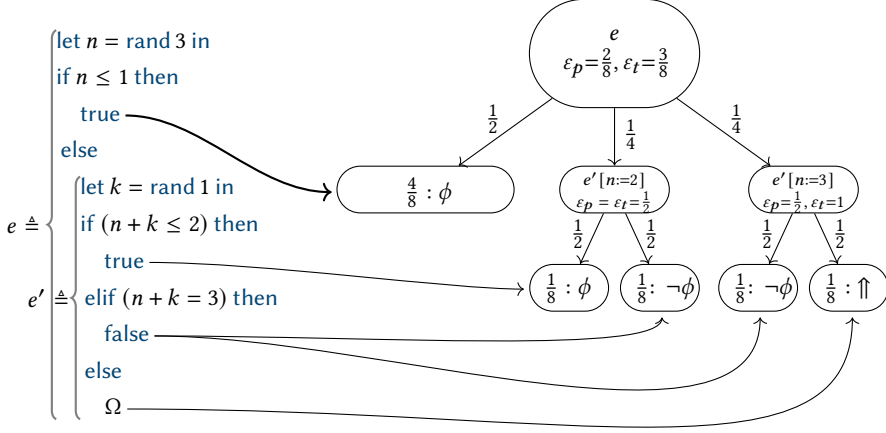
Fig. 1. Expected error analysis, partial and total.

introduction from the resource rules for the error credits and the HT-BIND and HT-FRAME rules. These derived rules are shown below.

$$\frac{\vdash \{P * \mathcal{f}(\varepsilon_1)\}\, e\, \{v.Q\} \qquad \vdash \forall v.\, \{Q * \mathcal{f}(\varepsilon_2)\}\, K[v]\, \{R\}}{\vdash \{P * \mathcal{f}(\varepsilon_1 + \varepsilon_2)\}\, K[e]\, \{R\}} \quad \text{HT-BIND-SIMPL}$$

$$\frac{\vdash \{P * \mathcal{f}(\varepsilon_1)\}\, e_1\, \{x. \mathcal{f}(\mathcal{E}_2(x)) * Q\} \qquad \vdash \forall x.\, \{Q * \mathcal{f}(\mathcal{E}_2(x))\}\, e_2\, \{R\}}{\vdash \{P * \mathcal{f}(\varepsilon_1)\}\, \text{let } x = e_1 \text{ in } e_2\, \{R\}} \quad \text{HT-BIND-EXP}$$

Proof rules for sampling. The only rules that make direct use of error credits are our novel rules involving sampling. 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 rule:

$$\frac{}{\vdash \{\mathcal{f}(\text{length}(xs)/(N+1))\}\, \text{rand } N\, \{n\, .\, n \notin xs\}} \quad \text{HT-RAND-ERR-LIST}$$

As mentioned in the introduction, Eris includes a more expressive 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 this 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 understand this rule in reverse: 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:

$$\frac{\sum_{i=0}^{N} \frac{\mathcal{E}_2(i)}{N+1} = \varepsilon_1}{\vdash \{\mathcal{f}(\varepsilon_1)\}\, \text{rand } N\, \{n\, .\, \mathcal{f}(\mathcal{E}_2(n))\}} \quad \text{RAND-EXP}$$

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 $n + k \le 2$

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 we can 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:

$$\left\{ \pounds \left(\tfrac{1}{4} \right) \right\} e \ \{\phi\}$$

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

$$\left\{ (n = 3 \vee n = 2) * \pounds \left(\tfrac{1}{2} \right) \right\} e' \ \{\phi\}$$

This Hoare triple is not difficult to prove; during the rand 1 step, we use the HT-RAND-ERR-LIST rule to spend $\pounds \left(\tfrac{1}{2} \right)$ 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 to n a random value, 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:

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

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

To summarise, 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 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 show how to do *amortized error* reasoning which, analogously to amortized running time bounds, assigns a uniform error cost to every operation on the data structure despite the real error cost increasing 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. In this data structure, we assume that on initialization we will allocate a block of memory 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 of memory. 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 will consider a faulty memory allocator, which has a small probability ε of failing on each write operation. Specifically, the allocator offers two methods extend and store with specifications:

$$\{ \text{\textsterling}\,(n \cdot \varepsilon) \, * \, l \mapsto^* vs \} \text{ extend } n \, l \, \{l'.\, l' \mapsto^* (vs +\!\!+ \text{replicate}(n, ()))\}$$

$$\{ \text{\textsterling}\,(\varepsilon) \, * \, l \mapsto^* vs * n < \text{length}(vs) \} \text{ store } l \, n \, v \, \{l \mapsto^* (vs[n := v])\}$$

Here we use $\mapsto^* vs$ to denote ownership of a points-to connective to each element of the list vs. Provided that 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 unit) appended after it. This incurs an error cost of $\text{\textsterling}\,(n \cdot \varepsilon)$. 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 $\text{\textsterling}\,(\varepsilon)$. 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:

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

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 is as follows:

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

Here, vec_spec vec vs should be read as "vec is a vector containing the values vs". Internally, the representation predicates 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 spare credits. Altogether, this will be enough to pay for the next resizing, which has cost $r \cdot \varepsilon$.

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

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

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 $\text{\textsterling}\,(3 \cdot \varepsilon)$ into $\text{\textsterling}\,(\varepsilon) \, * \, \text{\textsterling}\,(2 \cdot \varepsilon)$, using the first ε credits to pay for the call to store. From the definition of the representation predicate we get $\text{\textsterling}\,(p)$, 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 $\text{\textsterling}\,(2 \cdot \varepsilon + p)$ 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 $\mathcal{\xi}(p) * \mathcal{\xi}(2 \cdot \varepsilon)$, which we use to pay for the extend operation with credit $\mathcal{\xi}(r \cdot \varepsilon)$. 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 this model using a mutable map lm, which will serve as a cache of the hashes computed so far. If a key k has already been hashed we return the hash value stored in $lm(k)$. Otherwise, we sample a value uniformly from $V = \{0, \ldots, n\}$, store the value in $lm(k)$, and return it.

$$
\begin{aligned}
\text{compute\_hash } lm\, v \triangleq\ &\text{match get } lm\, v \text{ with} \\
&\quad \text{Some}(b) \Rightarrow b \\
&\quad |\ \text{None} \quad \Rightarrow \text{let } b = \text{rand } n \text{ in} \\
&\quad\qquad\qquad\qquad \text{set } lm\, v\, b; \\
&\quad\qquad\qquad\qquad b \\
&\text{end}
\end{aligned}
$$

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 can 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 more specific, suppose we have queried $f \triangleq \text{compute\_hash } lm$ 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 $\mathcal{\xi}\left(\frac{s}{n+1}\right)$ when choosing the new hash. We can encode this as a specification for our hash function in Eris:

$$\left\{ n \notin \text{dom } m * cf\_hashfun\, lm\, m\, V * \mathcal{\xi}\left(\tfrac{\text{size}(m)}{n+1}\right) \right\} f\, n\, \{v.\ cf\_hashfun\, lm\, (m[n \leftarrow v])\, V\}$$

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 $\sum_{i=0}^{MAX-1} \frac{i}{n+1} = \frac{(MAX-1)*MAX}{2(n+1)}$. We will require that the client always incurs the mean error $\frac{(MAX-1)*MAX}{2(n+1)*MAX} = \frac{(MAX-1)}{2(n+1)}$, which we denote as ε_{MAX}. Updating our specification,

$$\left\{ \begin{array}{l} size(m) < MAX * n \notin dom\ m\ * \\ amort\_cf\_hashfun\ lm\ m\ V\ *\ \maltese\,(\varepsilon_{MAX}) \end{array} \right\} fn\ \{v.\ amort\_cf\_hashfun\ lm\ m[n \leftarrow v]\ V\}$$

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 ε_{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*.[3]

The code for querying the hash function is shown below:

$$hash\_rs\ hf\ w \triangleq \mathsf{let}\ (lm, v, s, r) = hf\ \mathsf{in}$$

```
match get m 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
```

---

[3]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.

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:

$$cf\_hash\_rs \; hf \; m \; v \; s \; r \triangleq \exists lm, p. \; hf = (lm, v, s, r) * \not\xi \, (p) \tag{4}$$

$$p + (rval - sval) \cdot ((3 \cdot R_0)/(4 \cdot V_0)) \geq \sum_{i=s}^{r-1} i/v * \tag{5}$$

$$cf\_hashfun \; lm \; m \; v \; * (\dots) \tag{6}$$

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 $\not\xi \, (s/v)$. 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 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:

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

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 $\not\xi \, (s/v)$. From the precondition we have $\not\xi \, ((3 \cdot r_0)/(4 \cdot v_0))$, and from the representation predicate we can get the reserve $\not\xi \, (p)$. We can derive that this is enough to pay for $\not\xi \, (s/v)$ from condition (5) by an uninteresting, albeit nontrivial, calculation. Since the code branches depending on whether $s + 1$ is equal to r, we have the following specification:

$$\left\{ m[w] = \text{None} \; * (s + 1 \leq r) \; * cf\_hash\_rs \; f \; m \; v \; s \; r \; * \not\xi \left(\frac{3r_0}{4v_0} \right) \right\}$$

$$\text{hash\_rs} \; f \; w$$

$$\left\{ (b, f'). \; \begin{array}{l} (s + 1 < r * cf\_hash\_rs \; f' \; m[w := b] \; v \; (s + 1) \; r) \; \vee \\ (s + 1 = r * cf\_hash\_rs \; f' \; m[w := b] \; (2 \cdot v) \; (s + 1) \; (2 \cdot r)) \end{array} \right\}$$

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 (5) 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 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 table 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 for inserting a value w into a hash map hm:

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

The representation predicate isHashmap $hm \ ns$ should be understood as "hm is 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 if we own $\oint ((3 \cdot R_0)/(4 \cdot V_0))$, then insertion of an element w will always succeed. There are two cases in proving this speciffication: 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 $\oint ((3 \cdot R_0)/(4 \cdot V_0))$ to sample a fresh value from the hash map 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 table does, this establishes our specification no matter how many insertions have been performed before.

4.5 Amortised Hash Functions and Merkle Trees

A *Merkle tree* [Merkle 1988] 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 $\maltese\,(\varepsilon_{\mathsf{MAX}} \cdot \mathit{height}(\mathit{tree}))$ 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 is implemented using the hash_path helper function, which recursively computes the potential root hash from the input proof and leaf value:

$$
\begin{aligned}
&\mathsf{hash\_path\ f\ lproof\ lleaf} \triangleq \\
&\quad \mathsf{match\ lproof\ with} \\
&\quad\ (\mathsf{hd} :: \mathsf{tl}) \Rightarrow \mathsf{let\ (b, hash) = hd\ in} \\
&\qquad\qquad\qquad\quad \mathsf{if\ b} \\
&\qquad\qquad\qquad\quad \mathsf{then\ f\ ((hash\_path\ f\ tl\ lleaf) * 2^V + hash)} \\
&\qquad\qquad\qquad\quad \mathsf{else\ f\ (hash * 2^V + (hash\_path\ f\ tl\ lleaf))} \\
&\quad\ |\ \mathsf{nil} \qquad \Rightarrow \mathsf{f\ lleaf} \\
&\quad \mathsf{end}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{checker\ root\_hash\ f} \triangleq \\
&\quad \lambda \mathsf{lproof, lleaf.} \\
&\qquad \mathsf{let\ hp = hash\_path\ f\ lproof\ lleaf\ in} \\
&\qquad \mathsf{root\_hash == hp}
\end{aligned}
$$

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, 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:

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

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

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,
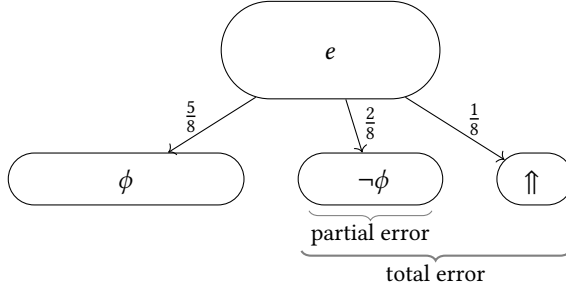
Fig. 2. Partial and total approximate correctness contrasted

    (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 $height(tree)$ 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 $height(tree)$ hashes computed by checker to preserve collision-freedom 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 spcification.

4.6 Futher Case Studies

For reasons of space, we omit other case studies, which can be found in the appendix. 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. The appendix also contains further details about the examples included in this section.

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 reconsider the example from Figure 1, which illustrates the distinction between partial and total correctness interpretations of approximate reasoning up-to-error. A summary of that example is depicted in Figure 2.

    In §3 we showed that in Eris we can prove the Hoare triple $\{ \maltese \left(\varepsilon_p \right) \} \, e \, \{x. x = \mathsf{true}\}$ for $\varepsilon_p = \frac{2}{8} = \frac{1}{4}$, intuitively because the program terminates with a result not satisfying the postcondition (false) with probability $\frac{1}{4}$. Note that the probability for 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 nontermination, and thus one needs $\varepsilon_t = \frac{3}{8}$ to show an approximate *total* Hoare triple $[\maltese \left(\varepsilon_t \right)] \, e \, [x. x = \mathsf{true}]$ (we use square 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 (precisely, $\varepsilon_t = 1$ 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 $[\mbox{\textit{\$}}(\varepsilon)] \; e \; [P]$ (for any postcondition P) in Eris_t can become vanishingly small, then e almost-surely terminates!

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. To reason about recursive programs in Eris_t we can instead apply the error credit rules we have developed thus far; we will see examples of how this works in the following.

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

THEOREM 8 (TOTAL ADEQUACY). *If* $\vdash [\mbox{\textit{\$}}(\varepsilon)] \; e \; [\phi]$ *then* $1 - \varepsilon \leq \mathrm{Pr}_{\mathrm{exec}\,(e,\sigma)}[\phi]$ *for any state* σ.

By a continuity argument in the meta logic (outlined in §6), we then obtain the following theorem.

THEOREM 9 (ALMOST-SURE TERMINATION). *If for all* $\varepsilon' > \varepsilon$ *we have* $\vdash [\mbox{\textit{\$}}(\varepsilon')] \; e \; [\phi]$, *then* $1 - \varepsilon \leq$ $\mathrm{Pr}_{\mathrm{exec}\,(e,\sigma)}[\phi]$ *for any state* σ.

If we pick $\varepsilon = 0$, this theorem allows us to conclude almost-sure termination of e by proving $\forall \varepsilon' > 0 . \; \vdash [\mbox{\textit{\$}}(\varepsilon')] \; e \; [\phi]$.[4] 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 $\mbox{\textit{\$}}(\varepsilon')$.

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. In the second case study we present a novel *planner* proof rule, which can separate credit arithmetic from concrete program steps, and show how to use it to prove correctness of general rejection samplers. Finally, we discuss other case studies we have done and which can be found in the appendix.

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 faulty 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 in a partial correctness logic that such algorithms only ever when they terminate (using HT-REC in Eris). However, it is more challenging to formally 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 uniform random samples, by strategically rejecting sequences which do not correspond to values in the target distribution. Consider the below implementation of a typical rejection sampling scheme $(S \; s \; c)$ with sampler program s and checker program c.

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

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

---

[4]To be precise, proving $\forall \varepsilon' > 0. \; \vdash [\mbox{\textit{\$}}(\varepsilon')] \; e \; [\phi]$ for any postcondition ϕ implies $\forall \varepsilon' > 0. \; \vdash [\mbox{\textit{\$}}(\varepsilon')] \; e \; [\top]$ by the rule of consequence. This allows us to conclude $\mathrm{Pr}_{\mathrm{exec}\,(e,\sigma)}[\top] = |\,\mathrm{exec}(e,\sigma)| = 1$ using Theorem 9.

Let us show that this uniform rejection sampler (S us_M uc_N) almost-surely terminates. Using Theorem 9, it suffices to show, for arbitrary nonnegative ε,

$$[\mathit{\sharp}(\varepsilon)] \; S \; \mathsf{us}_M \; \mathsf{uc}_N \; [\top]. \tag{7}$$

In proving this, how can we reason about the recursion in S? 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*. Note that using expectation-preserving composition, for any ε' we are able to prove

$$[\mathit{\sharp}(\varepsilon')] \; \mathsf{us}_M \; [v. \, \mathit{\sharp}(E(v) \cdot \varepsilon')] \qquad \text{where} \quad E(v) \triangleq \begin{cases} 0 & 0 \leq v \leq N \\ \frac{M+1}{M-N} & N < v \end{cases}$$

since $\frac{1}{M+1} \sum_{i=0}^{M} E(i) = 1$. In other words, each sampling attempt either produces a value the checker will certainly accept, or it multiplies our error credit by $\frac{M+1}{M-N} > 1$. 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 obtain a proof of False using $\mathit{\sharp}(1)$. Starting with any $\varepsilon > 0$, induction over $d(\varepsilon)$ allows us to prove (7), 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.

5.2.2 Presampling Tapes. In order to extract a general proof rule from our credit amplification argument, we will need to separate the credit accounting steps from the symbolic execution of a program. Because the error credit value 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 $\lambda_{\text{ref}}^{\text{rand}}$ 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 Gregersen et al. [2024], where tapes where originally introduced and used to sample in an asynchronous manner.

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

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

In addition to the heap, a state in $\lambda_{\text{ref}}^{\text{rand}}$ 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.

$$\frac{}{[\mathsf{True}] \; \mathsf{tape}(N) \; [\exists \iota. \, \iota \hookrightarrow (N, \epsilon)]} \; \text{ALLOC-TAPE}$$

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

$$\frac{}{[\iota \hookrightarrow (N, n \cdot \vec{n})] \ \mathsf{rand}\, N\, \iota\, [x.\, x = n \ast \iota \hookrightarrow (N, \vec{n})]} \ \text{\small LOAD-TAPE}$$

Note that there are no primitives in $\lambda_{\text{ref}}^{\text{rand}}$ 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.

$$\frac{[\iota \hookrightarrow (N, \vec{n} \cdot n)] \ e\ [P] \qquad 0 \le n \le N}{[\iota \hookrightarrow (N, \vec{n})] \ e\ [P]} \ \text{\small PRESAMPLE}$$

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.

$$\frac{\sum_{i=0}^{N} \frac{\mathcal{E}_2(i)}{N+1} = \varepsilon_1 \qquad \forall n.\, [\iota \hookrightarrow (N, \vec{n} \cdot n) \ast \mathit{\pounds}(\mathcal{E}_2(n))] \ e\ [P]}{[\iota \hookrightarrow (N, \vec{n}) \ast \mathit{\pounds}(\varepsilon_1)] \ e\ [P]} \ \text{\small PRESAMPLE-EXP}$$

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.

5.2.3 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 "planners" method by Pnueli and Zuck [2003]. Justified by the Borel-Cantelli lemma, Pnueli and Zuck establish a proof system whereby, to prove termination, one can reason as if a sequence of randomized choices will yield some prover-selected sequence of outputs infinitely often. Expressed in Eris$_t$, we have the following *planner* rule:

$$\frac{\begin{array}{cc} 0 < \varepsilon & \forall s.\, |z(s)| \le L \\ [\exists ys.\, \iota \hookrightarrow (N, xs +\!\!\!+ ys +\!\!\!+ z(xs +\!\!\!+ ys))] \ e\ [\phi] \end{array}}{[\iota \hookrightarrow (N, xs) \ast \mathit{\pounds}(\varepsilon)] \ e\ [\phi]} \ \text{\small PRESAMPLE-PLANNER}$$

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 "junk" samples ys in order to ensure it includes a target sequence $z(xs +\!\!\!+ ys)$ at the end. Generalizing Pnueli and Zuck'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 $xs +\!\!\!+ ys$ using a regular induction on lists, eventually ending up in their desired tape state $\iota \hookrightarrow (N, z(xs +\!\!\!+ ys))$.

The planner rule is derivable entirely within Eris$_t$ using the rules we have already seen. The proof proceeds using induction by credit amplification, and most of the proof is directly analogous to the argument in §5.2.1. The key change is in the following *amplification lemma*, which allows us to either sample a target word w onto a tape, or sample junk and multiply our error credit by a constant $k > 1$:

$$\frac{[\exists j.\, \iota \hookrightarrow (N, \vec{n} +\!\!\!+ j) \ast \mathit{\pounds}(k \cdot \varepsilon)] \ e\ [\phi] \qquad [\iota \hookrightarrow (N, \vec{n} +\!\!\!+ w)] \ e\ [\phi]}{[\iota \hookrightarrow (N, \vec{n}) \ast \mathit{\pounds}(\varepsilon)] \ e\ [\phi]}$$

Proving this involves $|z(\vec{n})|$ applications of PRESAMPLE-EXP. We provide the details, as well as the amplification constant k, in Appendix E.

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. We instantiate the rejection sampler S with the following sampler and checker:

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

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

$$[\sharp(\varepsilon)\ * \ 0 < \varepsilon * l \mapsto 0]\ S\ \mathsf{cs}_l\ \mathsf{cp}_l\ [\top].$$

Starting with any tape $\iota \hookrightarrow (N, xs)$, we invoke the planner rule with $\sharp(\varepsilon)$ and the target function

$$z(s) = \begin{cases} [1,1] & |s| \text{ is even}, \\ [0,1,1] & \text{otherwise}. \end{cases} \tag{8}$$

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 junk section of the tape, with each invocation of cs_l pulling off the two samples. Either the junk section will happen to contain some spurious $[1,1]$ sample, in which case the program terminates, or we step through the entire junk 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 9, as our initial error credit was arbitrarily small.

5.2.4 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 Appendix F 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

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

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. A complete definition with update modalities can be found in Appendix A.1.

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

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 $\not\xi(\varepsilon)$ through the *error interpretation* $\not\xi_\bullet(\varepsilon_1)$. Just like for the state interpretation, the error interpretation $\not\xi_\bullet(\varepsilon_1)$ connects the logical connective for error credits $\not\xi(\varepsilon)$ 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 $\not\xi(\varepsilon)$ asserts ownership of a fragmental element of the resource algebra, while $\not\xi_\bullet(\varepsilon)$ 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 camera.

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 $\mathrm{GLM}(e_1, \sigma_1, \varepsilon, Z)$. 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 simplied version is specified by the following rule (which should be read as a definition, expressing that the $\mathrm{GLM}(e_1, \sigma_1, \varepsilon, Z)$ predicate in the conclusion holds if the separating conjunction

of the premises above the line hold):

$$\frac{\text{red}(e_1, \sigma_1) \quad \varepsilon_1 + \varepsilon_2 \leq \varepsilon \quad \text{PGL}^{\varepsilon_1}_{\text{step}(e_1, \sigma_1)}[R] \quad \forall e_2, \sigma_2. R(e_2, \sigma_2) \twoheadrightarrow Z(e_2, \sigma_2, \varepsilon_2)}{\text{GLM}(e_1, \sigma_1, \varepsilon, Z)} \text{ STEP-SIMPLE}$$

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 so long as 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 use less error credit than was originally allotted. The user of the rule then picks an auxiliary intermediate predicate on configurations R. The premise $\text{PGL}^{\varepsilon_1}_{\text{step}(e_1, \sigma_1)}[R]$ 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. $\text{Pr}_{\text{step}(e_1, \sigma_1)}[\neg R] \leq \varepsilon_1$. 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 realised via the next rule.

$$\text{red}(\rho_1) \qquad \text{PGL}^{\varepsilon_1}_{\text{step}(\rho_1)}[R] \qquad \exists r. \forall \rho_2. \mathcal{E}_2(\rho_2) \leq r$$

$$\varepsilon_1 + \sum_{\rho_2 \in Cfg} \text{step}(\rho_1)(\rho_2) \cdot \mathcal{E}_2(\rho_2) \leq \varepsilon$$

$$\frac{\forall \rho_2. R(e_2, \sigma_2) \twoheadrightarrow \mathcal{E}_2(\rho_2) \geq 1 \vee Z(\rho_2, (\mathcal{E}_2(\rho_2)))}{\text{GLM}(\rho_1, \varepsilon, Z)} \text{ STEP-EXP}$$

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 $\text{step}(\rho_1)$. 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 10 (LIMIT WP ADEQUACY). *If* $\mathcal{E}(\varepsilon) \vdash \text{wp } e \{\phi\}$ *then* $\forall \sigma. \text{PGL}^{\varepsilon}_{\text{exec}(e, \sigma)}[\phi]$

Since exec is continuous in the sense that $(\forall n. \text{Pr}_{\text{exec}_n(e, \sigma)}[\phi] \leq x) \implies \text{Pr}_{\text{exec}(e, \sigma)}[\phi] \leq x$, 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 $\vdash \triangleright^n \mathrm{PGL}^{\varepsilon}_{\mathrm{exec}_n(e,\sigma)}[\phi]$ holds. The proof then proceeds by induction on the step index n. The inductive step for this argument hinges on the following lemma:

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

Intuitively, this says that the graded lifting modality can be composed with the partial graded lifting of the n-step execution of a program ρ to obtain a partial graded lifting of the execution of ρ for $n + 1$ steps. This should come as no surprise, since GLM internally uses PGL. The key lemma that allows this composition is then the corresponding composition lemma for partial graded liftings:

LEMMA 11. *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 $\mathrm{PGL}^{\varepsilon}_{\mu}[\phi]$ and $\exists x. \forall a. 0 \leq \mathcal{E}_2(a) \leq x$ and $(\forall a. \phi(a) \implies \mathrm{PGL}^{\mathcal{E}_2(a)}_{f(a)}[\psi])$, then $\mathrm{PGL}^{\varepsilon'}_{\mu \gg_f}[\psi]$, where $\varepsilon' = \varepsilon_1 + \sum_{a \in A} \mu(a) \cdot \mathcal{E}_2(a)$.*

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 9) 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

Approximate correctness. 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 composition, 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 where 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 functions. 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 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.

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.

Resource reasoning with credits. Resourceful reasoning about time and space complexity in separation logic via credits was pioneered by Atkey [2011]. The 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].

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.

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 have focused 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 such as Bao et al. [2022] and Li et al. [2023]. These works also focus on first-order programs. The concrete connection to our separation logic, where separating conjunction has the usual meaning, but specifications have a probabilistic interpretation is unclear, but an interesting object of future study.

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

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. *Proceedings of the ACM on Programming Languages* 5, ICFP (Aug. 2021), 93:1–93:30. https://doi.org/10.1145/3473598

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. *Proceedings of the ACM on Programming Languages* 4, POPL (Jan. 2020), 1–30. https://doi.org/10.1145/3371123 arXiv: 1907.10708.

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 *ACM Conference on Computer and Communications Security*. 62–73.

Juan Benet. 2014. IPFS - Content Addressed, Versioned, P2P File System. arXiv:1407.3561 [cs.NI]

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

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

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 *ACM Symposium on Operating System Principles*. https://www.amazon.science/publications/dynamo-amazons-highly-available-key-value-store

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, 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. *SIGPLAN Not.* 52, 1 (jan 2017), 205–217. https://doi.org/10.1145/3093333.3009855

John M. Li, Amal Ahmed, and Steven Holtzen. 2023. Lilac: A Modal Separation Logic for Conditional Probability. *Proc. ACM Program. Lang.* 7, PLDI, Article 112 (jun 2023), 24 pages. https://doi.org/10.1145/3591226

Ralph C. Merkle. 1988. A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology — CRYPTO '87*, Carl Pomerance (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 369–378.

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

C.H. Papadimitriou. 1991. On selecting a satisfying truth assignment. In *[1991] Proceedings 32nd Annual Symposium of Foundations of Computer Science*. 163–169. https://doi.org/10.1109/SFCS.1991.185365

Amir Pnueli and Lenore Zuck. 2003. Parameterized Verification by Probabilistic Abstraction. In *Foundations of Software Science and Computation Structures*, Andrew D. Gordon (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 87–102.

François Pottier, Armaël Guéneau, Jacques-Henri Jourdan, and Glen Mével. 2024. Thunks and Debits in Separation Logic with Time Credits. *Proceedings of the ACM on Programming Languages* 8, POPL (Jan. 2024), 50:1482–50:1508. 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

Calvin Smith, Justin Hsu, and Aws Albarghouthi. 2019. Trace abstraction modulo probability. *Proc. ACM Program. Lang.* 3, POPL, Article 39 (jan 2019), 31 pages. 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

A 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.

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

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 realised 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 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:

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

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.

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

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.

$$\mathsf{red}(e_1, \sigma_1) \qquad \mathrm{PGL}^{\varepsilon_1}_{\mathsf{state\_step}_\iota(\sigma_1)}[R] \qquad \exists r. \forall \rho_2. \mathcal{E}_2(\rho_2) \le r$$

$$\varepsilon_1 + \sum_{\sigma_2 \in State} \mathsf{state\_step}(e_1, \sigma_1)(e_1, \sigma_2) \cdot \mathcal{E}_2(e_1, \sigma_2) \le \varepsilon$$

$$\frac{\forall \sigma_2. R(\sigma_2) \twoheadrightarrow \mathcal{E}_2(e_1, \sigma_2) \ge 1 \lor \mathrm{GLM}(e_1, \sigma_2, \mathcal{E}_2(e_1, \sigma_2))}{\mathrm{GLM}(e_1, \sigma_1, \varepsilon, Z)} \text{ \textsc{statestep-exp}}$$

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

B   A SPECIFICATION FOR MAP

Consider the higher-order List.map function, which takes a function fv and a list lv as input, and returns a list where fv is applied to each element of the original list. Ignoring all error bounds reasoning, a reasonable specification for List.map could look something like the following (here, $list.map$ is a Coq-level map):

$$\left\{ \begin{array}{c} isList \; xs \; l \; * \\ \forall x. \{P(x)\} \; e \; x \; \{x'. \; x' = f(x) * Q(x')\} \; * \\ \ast_{x \in xs} P(x) \end{array} \right\} \text{List.map } e \; l \left\{ l'. \begin{array}{c} isList \; (list.map \; f \; xs) \; l' \; * \\ \ast_{x' \in (list.map \; f \; xs)} \; Q(x') \end{array} \right\} \tag{9}$$

Now consider the case where each application of fv incurs $\mathcal{E}(x)$ error credit, varying for each x stored in the list. It follows that we would want the term List.map $fv \; lv$ to incur exactly $\sum_{x \in l} \mathcal{E}(x)$ error credits, however if we attempt to prove the error-incurring specification by annotating Hoare triples with an "error budget" similar to the style of aHL Barthe et al. [2016b] we come across a few difficulties:

(1) By default, the logic itself does not allow the error to *depend on program terms*, i.e. in our example, the term l. We could circumvent this issue by forcibly adding a parameter representing the elements of l, but this involves a quantification in the metalogic which can make our specification harder to reuse.

(2) Even though the proof of an approximated specification follows a similar structure to the original, we cannot *reuse* the original specification to prove the approximate one as they live in different logics. One could retroactively reimplement the non-approximate specification in aHL using an error annotation of 0, but this is not a good strategy in general as it creates extra overhead and exposes unnecessary details to the proof engineer.

Instead, using the fact that error in Eris is an ordinary separation logic resource, we can easily verify an approximated specification by instantiating the predicates in 9 to include error credits:

$$\left\{ \begin{array}{c} isList \; xs \; l \; * \\ \forall x. \{P(x) * \maltese(\mathcal{E}(x))\} \; e \; x \; \{x'. \; x' = f(x) * Q(x')\} \; * \\ \ast_{x \in xs} \left(P(x) * \maltese(\mathcal{E}(x)) \right) \end{array} \right\} \text{List.map } e \; l \left\{ l'. \begin{array}{c} isList \; (list.map \; f \; xs) \; l' \; * \\ \ast_{x' \in (list.map \; f \; xs)} \; Q(x') \end{array} \right\} \tag{10}$$

We do not run into the same issues as our earlier example. By representing error as a resource, Eris can define flexible, value-dependent credits $\maltese(\mathcal{E}(x))$ without appealing to the metalogic. Moreover this error-incurring specification is a corrolary of the original non-approximate specification, facilitating more modular proofs which avoid proving redundant lemmas.

We mention that for a more coarse grained application, one can define the weaker specification where the total error credit for the map function is $\maltese(\mathrm{length}(l) * \max_{x \in l} \mathcal{E}(x))$. This specification is typical of approximate logics where value-dependent error is difficult to express, and follows from 10 by credit weakening.

C AMORTIZED HASH FUNCTIONS AND MERKLE TREES

In this case study, we highlight three striking aspects of Eris: the support for higher-order specifications, modularity reasoning, and amortized error reasoning. The example builds up through three levels of abstraction:

(1) we implement an amortized non-resizing hash function from a non-amortized one,
(2) we implement a Merkle tree library with the amortized hash function, and
(3) we apply the Merkle tree library to validate data stored in some unreliable storage.

C.1 Non-amortized non-resizing hash function

We first implement a 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 behaves as if for each key k, the hash $h(K)$ is randomly sampled uniformly and independently over V. We can implement such a hash function using mutable map m: to hash each key k we first check the map to determine whether it has been hashed before and if so, we return the hash value stored in $m(k)$. Otherwise, we randomly sample a value from $V = \{0, \ldots, n\}$, store the value in $m(k)$, and return it.

$$
\begin{aligned}
\text{compute\_hash } m\ v \triangleq\ &\text{match get } m\ v \text{ with} \\
&\quad \text{Some}(b) \Rightarrow b \\
&\quad |\ \text{None} \quad \Rightarrow \text{let } b = \text{rand } n \text{ in} \\
&\qquad\qquad\qquad\quad \text{set } m\ v\ b; \\
&\qquad\qquad\qquad\quad b \\
&\text{end}
\end{aligned}
$$

In the analysis of data structrures which use hash maps, one often assumes that a hash function is *collision-free*, in the sense that for a finite number of queries to the hash function with unique inputs, the output will never repeat. As written, this is *impossible*; the probability of a hash collision increases with the number of keys hashed, and in the extreme case where we query the hash function more than $n + 1$ times the pigeonhole principle ensures a collision must occur.

Nevertheless, this is not an issue in practice because the size of V is usually many magnitudes larger than the number of queries to the hash function. Therefore, it is the case that the hash function remains collision-free up to some small error. To be more concrete, consider the state where we have queried the hash function $f \triangleq \text{compute\_hash } m$ a total of s times each with a distinct input (so the map is now of size s). Suppose also that no query has produced a hash collision. Now to hash a fresh key without a hash collision, the hash function must "avoid" sampling from any of its prior s outputs. We can specify this in Eris by requiring that a query to f pay $\mathbf{\xi}\left(\frac{s}{n+1}\right)$:

$$
\left\{
\begin{array}{c}
n \notin \text{dom } m\ * \\
cf\_hashfun\ f\ m\ * \\
\mathbf{\xi}\left(\frac{\text{size}(m)}{n+1}\right)
\end{array}
\right\}\ f\ n\ \{v.cf\_hashfun\ f\ (m[n \leftarrow v])\}
$$

C.2 Amortized non-resizing hash function

One limitation of the above specification is that the error for each hash operation is proportional to the size of the map. This can complicate the error credit reasoning, particularly when many different clients perform a sequence of consecutive hash operations. Ideally we would want to amortize the error credits over a fixed number of hash queries MAX so that for each query, the requisite error is a fixed constant that does not depend on the state of inner map. We will realize this using the Eris interpretation of errors credits.

Starting from an empty map, if we bound the number of queries to be MAX the total number of errors used is $\frac{\xi}{}\left(\sum_{i=0}^{MAX-1}\frac{i}{n+1}\right) = \frac{\xi}{}\left(\frac{(MAX-1)*MAX}{2(n+1)}\right)$. We can evenly ammortize these errors across the sequence of queries so that each query pays $\frac{\xi}{}\left(\frac{(MAX-1)*MAX}{2(n+1)*MAX}\right) = \frac{\xi}{}\left(\frac{(MAX-1)}{2(n+1)}\right) \triangleq \frac{\xi}{}(\varepsilon_{MAX})$. Using this strategy, we can prove the following improved specification for our hash function:

$$\left\{\begin{array}{c} size(m) < MAX *\\ n \notin dom\ m *\\ amort\_cf\_hashfun\ f\ m *\\ \frac{\xi}{}(\varepsilon_{MAX}) \end{array}\right\}\ f\,n\ \{v.amort\_cf\_hashfun\ f\ m[n \leftarrow v]\}$$

Our proof reuses the original non-amortized specification by using the "piggy bank" method; the abstract predicate *amort_cf_hashfun* not only contains the *cf_hashfun* resource, but also any the error credits paid in excess during the first MAX/2 hash operations. That is, the first half of the queries store their extra credit into the "piggy bank", which the latter operations can use to pay for their more expensive queries.

C.3 Validating Data from an Unreliable Source

We tie up this case study by presenting an application of the Merkle tree library to validating of data stored in an unreliable source. By *unreliable*, we mean that we do not have any guarantees on the correctness of the "read" and "write" operations, meaning that after reading a client must also validate that their data has been stored faithfully.

Let *lis* be a list of naturals that we want to store in the unreliable storage. We will represent *lis* using a Merkle tree, where the elements of the list constitute the leaves. To implement a "write" operation we compute the hashes for the Merkle tree top to bottom and then store the entire tree into the unreliable storage, returning a reference to the merkle tree and a closure which remembers the root hash of the tree. This closure is essentially the checker from the aforementioned Merkle tree library, and we will use it to validate that the tree has correctly stored. To "read" an element from the list, we recursively walk down the Merkle tree in the unreliable storage, obtaining the final leaf node and a potential proof of its correctness. This information is then passed to the checker to determine whether the computed root hash coincides with the real root hash value. In total, applying our Merkle tree library each read operation requires an error credit of $\frac{\xi}{}(\varepsilon_{MAX} * height(tree))$.

> *leaf_lookup loc height idx* checker $\triangleq$
>
> > let (lproof, lleaf) = *read_tree loc height idx* in
> > if *bounds_check* lproof lleaf then
> > > if checker lproof lleaf
> > > > then Some lleaf
> > > > else None
> > else None

Here the *read_tree* program takes the location of the root of the tree, the height, and the index of the leaf node we are reading, and returns a tuple containing the proof and the leaf value. The function *bounds_check* verifies that the proof and the leaf value lie within expected bounds. It is worth noting that the implementation and correctness guarantees of *read_tree* and *bounds_check* are relatively unimportant to the specification of *leaf_lookup*, as our specification guarantees that we return Some *x* only if *x* is the true value of the *idx*-th leaf value of the Merkle tree regardless.

Altogether, to correctly execute *leaf_lookup l* (*height(tree)*) *idx* checker for some location *l* in unreliable storage we require the following preconditions:

(1) The number idx is smaller than $2^{height(tree)}$.
(2) The function checker from the merkle tree library is specialized to both $tree$ and m.
(3) The merkle tree $tree$ is built correctly from hash map m and leaves lis.
(4) The function f encodes the amortized hash function under the map m.
(5) The size of m plus the height of the tree is smaller or equals to MAX.
(6) At least enough error credit for the read.

Expressed in Eris,

$$
\left\{
\begin{array}{c}
idx < 2^{height(tree)} \; * \\
checker\_spec \text{ checker } tree \; m \; * \\
tree\_valid\_with\_leaf\_list \text{ } tree \text{ } lis \text{ } m \; * \\
amort\_cf\_hashfun \text{ } f \text{ } m \; * \\
size(m) + height(tree) <= \text{MAX} \; * \\
\not{\xi}\,(\varepsilon_{\text{MAX}} * height(tree))
\end{array}
\right\}
$$

$$leaf\_lookup \; l \; (height(tree)) \; idx \text{ checker}$$

$$\{v.v = \text{Some } x \implies lis[idx] = x\}$$

For a similar reason as to how we do not impose many guarantees on the functions $read\_tree$ and $bounds\_check$, nowhere in the specification do we impose any restrictions on the location l. In particular, we do not even know that l is a reference to the correct Merkle tree in the unreliable storage as we do not assume any behavior of the unreliable storage.

D AMORTIZED ERROR FOR COLLISION-FREE HASH MAP

Hash maps are one of the most ubiquitous data structures in programming, since they can represent large sets with efficient insertion, deletion and lookup operations. This 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, thus 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 collision happening. However, computing this probability over a sequence of insertions is cumbersome, due to its dependence on the current size of the hash table and the number of elements it contains. Moreover, it also leads to less modular specifications for programs that use hash maps as their components.

We will use the dynamically-resizing hash function defined in the main body of the paper to implement a collision-free dynamically-resizing hash map, with amortized cost of insertion. We will have 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 $2 * v$ and we set r to $2 * r$. The code is shown below:

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

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 $\maltese((3 \cdot R_0)/(4 \cdot V_0))$. We will now explain how we achieve this. First consider the representation predicate for the hash map below:

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

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. Crucially, this predicate keeps no track of error credits, all of the error accounting is done through the cf_hash_rs predicate, which is used as a client. With this representation predicate, we can prove the following specification for $insert$.

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

This specification states that if we own $\maltese((3 \cdot R_0)/(4 \cdot V_0))$, then insertion of an element w will always succeed. There are two ways in which this can be validated. Either w was already in the hash map (and therefore $ns = ns \cup \{w\}$) or it is a new element. If it is a new element, we also have to case on whether we have to resize or not. In either of those cases, we can use $\maltese((3 \cdot R_0)/(4 \cdot V_0))$ to sample a fresh value from the hash map, following the specifications proven for the resizing hash function. This ensures that the location in the table corresponding to that index is uninitialized. Furthermore, by ensuring that the hash map resizes at the same time as the hash table does, this specification will be valid no matter how many insertions have been performed before.

Notice here how the modularity of Eris enables us to have a relatively simple specification, which besides the presence of error credits, coincides with the specification one would expect for a hash map.

E EXPECTATION-PRESERVING COMPOSITION ON WORDS

While 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 $0 \leq i \leq L$, define the constants

$$ecAmp_{N,L} \triangleq 1 + \frac{1}{(N+1)^L - 1} \qquad\qquad ecRem_{N,L}(i) \triangleq 1 - \frac{(N+1)^i - 1}{(N+1)^L - 1}$$

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

$$D_{N,L}^\varepsilon(i, c) \triangleq \begin{cases} ecRem_{N,L}(i+1)\,\varepsilon & c = \vec{w}[i] \\ ecAmp_{N,L}\,\varepsilon & \text{otherwise} \end{cases}$$

Starting with $\sharp\left(ecRem_{N,L}(i)\,\varepsilon\right)$ the function $D_{N,L}^\varepsilon(i, \_)$ is mean-preserving, since

$$\sum_{c=0}^{N} D_{N,L}^\varepsilon(i, c) = ecRem_{N,L}(i+1)\,\varepsilon + N\,ecAmp_{N,L}\,\varepsilon$$

$$= \left(\frac{(N+1)(N+1)^L - (N+1)^{i+1}}{(N+1)^L - 1}\right)\varepsilon$$

$$= (N+1)\,ecRem_{N,L}(i)\,\varepsilon$$

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 ith sample we have will have correctly sampled the first i characters of $\vec{w}$ and own $\sharp\left(ecRem_{N,L}(i)\,\varepsilon\right)$. At step i, perform expectation preserving composition using the error function $D_{N,L}^\varepsilon(i, \_)$. Each composition either correctly samples the next character of $\vec{w}$ and decreases the error credit supply to $\sharp\left(ecRem_{N,L}(i+1)\,\varepsilon\right)$, or increases it to $\sharp\left(ecAmp_{N,L}\,\varepsilon\right)$. Note that $\sharp\left(ecRem_{N,L}(0)\,\varepsilon\right) = \sharp(\varepsilon)$ for the initial case, and $\sharp\left(ecRem_{N,L}(L)\,\varepsilon\right) = \sharp(0)$ 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 $ecAmp_{N,L}$.

Implemented using PRESAMPLE-EXP, this procedure proves the amplification lemma from §5.2.3:

$$\frac{\left[\exists \vec{j}.\, \iota \hookrightarrow (N, \vec{n} \mathbin{+\!\!+} \vec{j}) * \sharp\left(ecAmp_{N,L}\,\varepsilon\right)\right] e\,[\phi] \qquad [\iota \hookrightarrow (N, \vec{n} \mathbin{+\!\!+} \vec{w})]\,e\,[\phi]}{[\iota \hookrightarrow (N, \vec{n}) * \sharp(\varepsilon)]\,e\,[\phi]}$$

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 $\vec{w}$: $ecExc_{N,L} \triangleq ecAmp_{N,L} - 1$. Since $ecRem_{N,L}(i) < 1$ for all $0 \leq i \leq L$, we can prove that

$$\sharp\left(ecAmp_{N,L}\right) \mathbin{-\!\!*} \sharp\left(ecRem_{N,L}(i)\right) * \sharp\left(ecExc_{N,L}\right) \tag{11}$$

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 $\sharp\left(ecExc_{N,L}\,\varepsilon\right)$.

F   RANDOMIZED SAT SOLVING

Our *induction by error amplification* technique it not limited to proving termination for simple rejection samplers. To demonstrate the versatility of this approach, we develop a higher order specification which can prove termination properties of "check and retry" algorithms which have complex dependencies on state.

F.1   A Higher Order Specification for Rejection Samplers

Let s and c be a sampler/checker pair, and $\Theta : Val \rightarrow iProp$ be a property of samples. We seek to establish a relationship between s and c such that $(S\ s\ c)$ almost certainly terminates with a value satisfying Θ.

Recall that our formulation of the planner rule allows us to almost surely sample some target word onto a tape, and that the target word is free to depend on the state of the tape beforehand. Explicitly providing such a target word as a function of a sequence of sampling events may be unnecessarily cumbersome, and may not even be sufficient to execute the program, since the planner rule makes no assertions over the *junk* section sampled before the target word. Rather than explicitly specifying a target sampling event, it may be more straightforward to leave the target event implicit and specify how c behaves in terms of s directly.

$$sc(s, c, k, \Theta) \triangleq \ulcorner k > 1 \urcorner * \forall \varepsilon > 0.$$

$$[\text{\textsterling}(\epsilon)]\ s\ ()\ \left[v.\ \begin{array}{l} ([\text{True}]\ c\ v\ [r : \mathbb{B}.\ \ulcorner r = \text{true} \urcorner * \Theta\ v]) \vee \\ ([\text{True}]\ c\ v\ [r : \mathbb{B}.\ \ulcorner r = \text{true} \urcorner \mathbin{-\!\!*} \Theta\ v] * \text{\textsterling}(k\ \epsilon))) \end{array} \right] \quad (12)$$

Figure 12 leverages a higher-order specification in order to establish such a relationship.[5] The specification reads as follows: for a fixed constant k and any positive amount of error credit ε, we can execute the sampler to obtain one of two outcomes:

(1) a *productive sample*, which ensures the checker will step to true, or
(2) an *erroneous sample*, which does not guarantee the sampler will step to true, but amplifies our error credit by k.

Figure 12 additionally specifies that Θ holds whenever c accepts a sample, even if that sample is erroneous. Since we do not need an exact error bound to prove almost sure termination, this allows a prover to underapproximate the cases where c will accept.

We can perform induction by error amplification for any sampler and checker pair satisyfing this specification. A sequence of $d = \lceil \log_k(1/\epsilon) \rceil$ erroneous samples guarantees that we accumulate $\text{\textsterling}(1)$, so by induction on d we can obtain

$$(\exists k, sc(s, c, k, \Theta)) \mathbin{-\!\!*} \forall \varepsilon > 0,\ [\text{\textsterling}(\varepsilon)]\ S\ s\ c\ [r.\ \Theta\ r]. \quad (13)$$

When Θ is pure[6] our continuity result yields

$$(\exists k,\ sc(s, c, k, \Theta)) \rightarrow \forall \sigma.\ \text{Pr}_{\text{exec}\ (S\ s\ c, \sigma)}[\Theta] = 1.$$

F.2   An Incrementalized Specification

Next, we generalize specification 12 to allow the sampler to have effects which persist between attempts. This is necessary in order to support samplers whose "productive" guesses may not immediately cause the checker to accept, but rather make incremental progress towards an accepting state.

---

[5]In this section, we abbreviate $[\Phi]\ e\ [v.\ \ulcorner v = \text{true} \vee v = \text{false} \urcorner * \Psi]$ as $[\Phi]\ e\ [v : \mathbb{B}.\ \Psi]$.
[6]To prove termination alone, one can use $\Theta = (\lambda\_.\ \top)$, which is pure.

$$isc(s, c, P, E, L_p, L_e, \Theta) \triangleq$$
$$[\mathcal{f}(E(0)) \lor P(0)] \; s \; () \; [v. \, [\text{True}] \; c \; v \; [r : \mathbb{B}. \, \ulcorner r = \text{true} \urcorner * \Theta \, v]] \; *$$
$$\forall i < L_e, \; j < L_p.$$

$$\begin{bmatrix} \mathcal{f}(E(i+1)) \\ * \, P(j+1) \end{bmatrix} s \; () \left[r. \begin{bmatrix} [\text{True}] \; c \; v \; [r : \mathbb{B}. \, \ulcorner r = \text{true} \urcorner * \Theta \, v] \; \lor \\[2mm] [\text{True}] \; c \; v \left[r : \mathbb{B}. \begin{array}{l} \mathcal{f}(E(i+1)) \, * \, P(j) \, * \\ (\ulcorner r = \text{true} \urcorner \, \rightarrow\!\!* \; \Theta \, v) \end{array} \right] \; \lor \\[4mm] [\text{True}] \; c \; v \left[r : \mathbb{B}. \, \exists j' \le L_p. \begin{array}{l} \mathcal{f}(E(i)) \, * \, P(j') \, * \\ (\ulcorner r = \text{true} \urcorner \, \rightarrow\!\!* \; \Theta \, v) \end{array} \right] \end{bmatrix} \right]$$
$$(14)$$

This specification generalizes 12 by paramaterizing two key components. The first is $E : \mathbb{N} \to \mathbb{R}_{\ge 0}$, which quantifies the *error* in terms of the number of failed samples remaining before we can apply a credit spending argument. The specification also generalizes over the remaining *progress* towards an accepted sample via the term $P : \mathbb{N} \to iProp$. As a regular Iris proposition, P can contain invariants relating to program state and the resources required to execute s or c. Both E and P have fixed bounds on their argument L_e and L_p. typical instansiation sets $E(L_e) = 0$ so that we can start with $\mathcal{f}(E(L_e))$, and $E(0) = 1$ so $\mathcal{f}(E(0)) \rightarrow\!\!* \bot$.

We will now break down 14 in more detail. The first conjunct is a base case, which states that either complete error $\mathcal{f}(E(0))$ or complete progresss $P(0)$ are enough to step the sampler and checker to the desired result. Like the non-incremental specification, the second conjunct of 14 specifies the behavior of the checker in terms of the sampler's result. Namely, executing s yields one of three cases:

(1) a *lucky sample*, which allows the checker to terminate immediately,
(2) an *improvement sample*, which makes progress by decreasing the argument to P, or
(3) an *erroneous sample*, which loses progress but generates enough credit to decrease E.

We can prove a total specification for samplers and checkers which satisfy 14:

$$isc(s, c, P, E, L_p, L_e, \Theta) \rightarrow\!\!* \forall i \le L_e, \; j \le L_p. \; [E(i) * P(j)] \; S \; s \; c \; [r. \, \Theta \, r] \qquad (15)$$

The proof of 15 proceeds by a nested induction, firstly over the remaining erroneous samples, and secondly over the remaining progress samples. That is, the proof attempts to sample a sufficiently long sequence of improvement samples or improve the error at least once. Even through it is possible for the sampler to lose progress on P, it never loses the error credits in E, so this induction is well-founded.

F.3 Almost Sure Recognition in Randomized SAT Solving

WalkSAT [Papadimitriou 1991] is a randomized algorithm for recognizing satisfiable boolean formulas. Given a boolean formula in conjunctive normal form, WalkSAT searches for a solution by iteratively flipping random variables inside unsatisfied clauses. Importantly, the possible variables which WalkSAT might flip depends both on both the current assignment and also the formula under test. This means that, unlike the case for a simple random walk on $\mathbb{B}^N$, it is not immediate that WalkSAT almost-surely terminates even when the formula is satisfiable.[7] By proving an instance of 14 and applying a continuity argument, we will show that WalkSAT almost surely recognizes satisfiable boolean formulas.

---

[7]For example, a buggy implementation of WalkSAT which never resamples from the first clause will fail to recognize the satisfiable formula $(X_1 \lor X_1 \lor X_1) \land (X_2 \lor X_2 \lor X_2)$ if X_1 is False in its inital assignment.

$$eval\_var((n, p)) \triangleq \lambda a. \; \mathsf{let} \; b = \mathsf{eval\_asn} \; a \; n \; \mathsf{in}$$
$$\mathsf{match} \; p \; \mathsf{with}$$
$$Pos \; \Rightarrow b$$
$$| \; Neg \Rightarrow \sim b$$
$$\mathsf{end}$$

$$eval\_clause((x_0, x_1, x_2)) \triangleq \lambda a. \; eval\_var \; x_0 \; a \; \&\& \; eval\_var \; x_1 \; a \; \&\& \; eval\_var \; x_2 \; a$$

$$resample((n_0, \_), (n_1, \_), (n_2, \_)) \triangleq \lambda l. \; \mathsf{let} \; a = \; ! l \; \mathsf{in}$$
$$\mathsf{let} \; i = \mathsf{rand} \; \#2 \; \mathsf{in}$$
$$\mathsf{let} \; b = \mathsf{eval\_asn} \; a \; n_i \; \mathsf{in}$$
$$l \leftarrow \mathsf{upd\_asn} \; a \; n_i \; (\sim b)$$

$$sampler(f) \triangleq \lambda l. \; \mathsf{match} \; f \; \mathsf{with}$$
$$[] \quad\quad \Rightarrow ()$$
$$| \; (c : cs) \Rightarrow \mathsf{if} \; eval\_clause \; c \; (! l)$$
$$\mathsf{then} \; sampler \; cs \; l$$
$$\mathsf{else} \; resample \; c \; l$$
$$\mathsf{end}$$

$$checker(f) \triangleq \lambda l. \; \mathsf{match} \; f \; \mathsf{with}$$
$$[] \quad\quad \Rightarrow \mathsf{true}$$
$$| \; (c : cs) \Rightarrow eval\_clause \; c \; (! l) \; \&\& \; checker \; cs \; l$$
$$\mathsf{end}$$

$$W_f \triangleq \lambda l. \; S \; (sampler \; f \; l) \; (checker \; f \; l)$$

Fig. 3. A sampler and checker implementing a simplified version of WalkSAT.

Figure 3 depicts our implementation of WalkSAT as a sampler/checker pair. Our $\lambda_{\mathrm{ref}}^{\mathrm{rand}}$ program is paramaterized by the number of varibles N and a 3SAT formula f at the Coq-level. We represent a formula f as a list of *clauses*, each consisting of a triple of *variables*, comprised of a *variable index* $n < N$ and a *polarity*. As an example, we represent the 3SAT clause $(x_1 \vee x_2 \vee \overline{x_3})$ as the triple $((1, Pos), (2, Pos), (3, Neg))$.

We represent an *assignment* of variable indices to boolean values in $\lambda_{\mathrm{ref}}^{\mathrm{rand}}$ as a length N linked list, and in Coq as a length N list of booleans. The Coq-level predicate $SAT_f(a)$ means that a is a length N boolean list, and it satisfies the formula f. The proposition $inv\_asn \; v \; s$ asserts that the ProbLang value v represents the same assignment as the list s. The standard programs eval_asn and upd_asn look up and update values in a ($\lambda_{\mathrm{ref}}^{\mathrm{rand}}$ level) assignment by index, and the program init_asn allocates a new assignment with random values.

The functions *eval_var*, *eval_clause* and *checker* determine if the current $\lambda_{\mathrm{ref}}^{\mathrm{rand}}$ assignemnt satisfies a variable, clause, and formula respectively. Given a clause, the function *resample* chooses

a variable index referenced by that clause uniformly at random, and negates it in the current assignment. Finally, the function *sampler* resamples from the first unsatisfied clause, doing no such clauses exist. Combined, the program W_f implements a simplified[8] version of the WalkSAT algorithm: the program repeatedly resamples the first unsatisfied clause and terminates when the current assignemnt satisfies f.

Let us assume there is some solution t to a 3SAT formula f, that we own a positive amount of error credit $\oint(\varepsilon)$, and that we own an assignment l. Denote the Hamming distance between t and a boolean list of the same length as $d^{(0)}(a)$. Now, we can define the key components for an instance of 14:

$$L_p \triangleq N$$
$$L_e \triangleq \lceil 1/ecExc_{3,N} \rceil$$
$$P_W(n) \triangleq \exists a,\ m.\ (l \mapsto a) * \oint\left(ecRem_{3,N}(N - n)\,\varepsilon\right) * \ulcorner inv\_asn\ a\ m \urcorner * \ulcorner d^{(0)}(asn) \leq n \urcorner$$
$$E_W(n) \triangleq \max(0, 1 - n\ ecExc_{3,N}\ \varepsilon)$$
$$\Theta_W \triangleq \exists a,\ m.\ (l \mapsto a) * \ulcorner inv\_asn\ a\ m \urcorner * \ulcorner SAT_f\ m \urcorner$$

Our error measure E_W is a linear function of the minimum amplification of length N words in 3 characters. Our progress measure P_W asserts both ownership over a $\lambda_{\text{ref}}^{\text{rand}}$ level assignment, and bounds the Hamming distance between that assignment and t. Interestingly, the progress measure also contains an error term. Because correcting an assignment can take up to N lucky resamples, our amplification strategy requires some nonzero source of credit which is free to decrease as *improvement* samples are drawn. As a resouce, separating error credits to isolate such dependencies is no issue in Eris. Finally, the resulting proposition Θ_W asserts that the $\lambda_{\text{ref}}^{\text{rand}}$ level value corresponds to a satisfying assignment for f. Note that we are not proving that the final assignemnt matches t; for formulas with multiples solutions this is in fact not true.

Now we can outline the proof of $isc(sampler\ f\ l, checker\ f\ l, P_W, E_W, L_p, L_e, \Theta_W)$. The first conjunct is trivial, we must show that either $E_W(0)$ or $P_W(0)$ is enough for W_f to terminate and satsify Θ. The final error case is a straightforward credit spending argument, since $\oint(E_W(0)) = \oint(()\ 1)$. For the final progress case, the Hamming Distance criteeon in P_W ensures that the current state is t, which we know to be a solution to f as required.

The second conjuct pertains to the behavior of the sampler given some initial $\oint(E_W(i + 1))$ and $P_W(j + 1)$. If the intial state is some solution to f, the sampler will not resample any clause and we can finish by the *lucky sample* case. Otherwise, the current assignment is not a solution to f, meaning it must necessarily differ from t at some variable $0 \leq z \leq 2$ in its first unsatsified clause (or else that clause would be satisfied). The resources in $P_W(j + 1)$ are enough to step us up to the point where *resample* chooses which variable to flip, at which point we perform an expectation-preserving composition using the function $D_{3,N}(N - n, z)$ and the credits $\oint(ecRem_{3,N}(N - j)\,\varepsilon)$. The case where *resample* flips the zth variable decreases the Hamming distance between the current assignment and t by 1, meaning we can establish the *improvement sample* case. Otherwise, the program flips some other variable, and we will establish the *erroneous sample* case. This goal has two main parts: we must find some $j' \leq L_p$ such that prove $P_W(j')$, and we must improve our error measure from $\oint(E(i + 1))$ to $\oint(E(i))$. As remarked in section E, we can split our newly generated error credit $\oint\left(ecAmp_{3,N}\,\varepsilon\right)$ into $\oint(ecExc_{3,N}\,\varepsilon)$ and $\oint(ecRem_{3,N}(L_p))$. The excess $\oint(ecExc_{3,N}\,\varepsilon)$ credit is exactly the difference between $E(i + 1)$ and $E(i)$, allowing us to improve the error measure. The $\oint(ecRem_{3,N}(L_p))$ term is enough to re-establish $P_W(L_p)$ regardless of the effect the incorrect

---

[8]Implementations of WalkSAT typically choose a *random* unsatisfied clause, rather than the first one. This detail is irrelevant to our analysis.

flip had on the assignment, since the Hamming distance between any assignment and t is at most L_p. Therefore, our implementation WalkSAT satisfies our general incremental specification isc($sampler\ f\ l$, $checker\ f\ l$, P_W, E_W, L_p, L_e, Θ_W).

To conclude this section, we will show that WalkSAT almost surely terminates when the formula is satisfiable. We have shown that we can construct a *isc* instance for any initial postitive error credit ε, so by 15 we have

$$\forall i \leq L_e,\ j \leq L_p.\ [E(i) * P(j)]\ S\ s\ c\ [r.\Theta\,r]$$

Furthermore $ecRem_{3,N}(L_p) = E_W(L_e) = 0$, and $d^{(0)}(a) \leq L_p$ for all assignments a. This means that given $\pounds(\varepsilon)$ we can establish our intial conditions $\pounds(E_W(L_e))$ and $P_W(L_p)$ for any starting assignment:

$$\forall \epsilon > 0,\ \left[\pounds(\epsilon)\ *\ \ulcorner \exists t.\ SAT_f\ t \urcorner\right]\ \mathsf{let}\ l = \mathsf{init\_asn}\ \mathsf{in}\ (W_f\ l)\ \left[\exists v, a.\ l \mapsto v\ *\ \ulcorner inv\_asn\ v\ a \urcorner\ *\ \ulcorner SAT_f\ a \urcorner\right]$$
$$(16)$$

Finally, since 16 is quantified over ε, Theorem 9 allows us to conclude that WalkSAT almost surely terminates for a satisfiable formula.