Verifying Probabilistic Programs with Separation Logic Part 2

Joseph Tassarotti
New York University

Based on joint work with: Alejandro Aguirre, Philipp G. Haselwarter, Kwing Hei-Li, Markus de Medeiros, Puming Liu, Alex Bai, Simon Oddershede Gregersen, and Lars Birkedal

Course page: clutch-project.org/epit2026.html
Rocq code: github.com/logsem/clutch/tree/epit2026

Recap of Part 1

Goal: bridge probabilistic verification and modern separation logic.

Recap of Part 1

Goal: bridge probabilistic verification and modern separation logic.

Three approaches to probabilistic Hoare triples: distributional assertions, expectation transformers, and liftings.

Recap of Part 1

Goal: bridge probabilistic verification and modern separation logic.

Three approaches to probabilistic Hoare triples: distributional assertions, expectation transformers, and liftings.

Eris: error credits $\lightning(\epsilon)$ as a resource:

$$\{\lightning(\epsilon) * P\}\, e\, \{Q\} \quad \text{means } Q \text{ holds except with probability} \leq \epsilon$$

- Splittable: $\lightning(\epsilon_1 + \epsilon_2) \dashv\vdash \lightning(\epsilon_1) * \lightning(\epsilon_2)$, bounded by $\lightning(1) \vdash \mathrm{False}$
- Averaging rule supports redistributing credit across sample branches
- Resolves aHL limitations: error dependency, propagation, amortization

Example: Morris's Approximate Counter

Problem: count up to n events using fewer than $\log_2 n$ bits.

Example: Morris's Approximate Counter

Problem: count up to n events using fewer than $\log_2 n$ bits.

Idea (Morris, 1978): store an approximation of $\log_2 n$ rather than the count n itself. That only needs $O(\log_2 \log_2 n)$ bits.

Example: Morris's Approximate Counter

Problem: count up to n events using fewer than $\log_2 n$ bits.

Idea (Morris, 1978): store an approximation of $\log_2 n$ rather than the count n itself. That only needs $O(\log_2 \log_2 n)$ bits.

But how can you increment? Randomness!

Example: Morris's Approximate Counter

Problem: count up to n events using fewer than $\log_2 n$ bits.

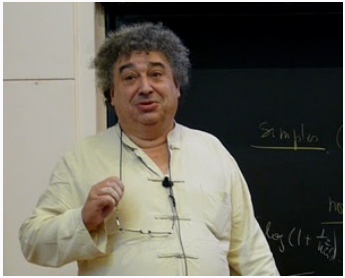
Idea (Morris, 1978): store an approximation of $\log_2 n$ rather than the count n itself. That only needs $O(\log_2 \log_2 n)$ bits.

But how can you increment? Randomness!

$$
\begin{aligned}
&\mathit{incr} \triangleq \lambda c. \\
&\quad \mathsf{let}\, x = !\, c\, \mathsf{in} \\
&\quad \mathsf{if}\ \mathsf{rand}(2^x - 1) = 0\, \mathsf{then}\ \ c \leftarrow x + 1\, \mathsf{else}\, () \\
&\mathit{read} \triangleq \lambda c.\ \ 2^{!\, c} - 1
\end{aligned}
$$

Key property: expected value of *read* c is n.

Philippe Flajolet (1948–2011)



- A pioneer of analytic combinatorics for analysis of algorithms.
- 1985 paper analyzed many deep properties of Morris's counter.

Philippe Flajolet (1948–2011)

- A pioneer of analytic combinatorics for analysis of algorithms.
- 1985 paper analyzed many deep properties of Morris's counter.

EPIT depuis 1973

1. Langages algébriques, Bonascre (J.-P. Crestin et M. Nivat), 1973
2. Complexité des algorithmes, Ile de Berder (Ph. Flajolet), 1974

Philippe Flajolet (1948-

A FIRST COURSE IN STOCHASTIC PROCESSES

SECOND EDITION

SAMUEL KARLIN
STANFORD UNIVERSITY AND THE WEIZMANN INSTITUTE OF SCIENCE

HOWARD M. TAYLOR
CORNELL UNIVERSITY

ACADEMIC PRESS New York San Francisco London
A Subsidiary of Harcourt Brace Jovanovich, Publishers

ombinatorics for analysis

any deep properties of

EPIT depuis 197

1. Langages alg t M. Nivat), 1973
2. Complexité Flajolet), 1974

Bounding the counter cell

In practice the cell holds a b-bit value, so we must rule out $x + 1 \geq 2^b$.

Bounding the counter cell

In practice the cell holds a b-bit value, so we must rule out $x + 1 \geq 2^b$.

Our language doesn't have bounded value, but we can simulate by adding an assertion that fails if the new exponent would overflow:

$$\begin{aligned} &\mathit{incr} \triangleq \lambda c.\ \ \mathsf{let}\ x = !\,c\ \mathsf{in} \\ &\quad \mathsf{if}\ \mathsf{rand}(2^x - 1) = 0\ \mathsf{then}\ \mathsf{assert}(x + 1 < 2^b);\ \ c \leftarrow x + 1\ \mathsf{else}\ () \end{aligned}$$

Bounding the counter cell

In practice the cell holds a b-bit value, so we must rule out $x + 1 \geq 2^b$.

Our language doesn't have bounded value, but we can simulate by adding an assertion that fails if the new exponent would overflow:

$$\begin{array}{l} \mathit{incr} \triangleq \lambda c.\ \ \mathsf{let}\ x = !\,c\ \mathsf{in} \\ \quad \mathsf{if}\ \mathsf{rand}(2^x - 1) = 0\ \mathsf{then}\ \mathsf{assert}(x + 1 < 2^b);\ \ c \leftarrow x + 1\ \mathsf{else}\ () \end{array}$$

A verified spec for *incr* must rule out the assertion failing.

Spec 1: a state-tracking specification

$$counter\_val(c, k) \triangleq c \mapsto k.$$

Spec 1: a state-tracking specification

$counter\_val(c, k) \triangleq c \mapsto k.$

$$\left\{counter\_val(c, k) * k + 1 < 2^b\right\}$$

$$incr\ c$$

$$\{counter\_val(c, k) \vee counter\_val(c, k + 1)\}$$

Spec 1: a state-tracking specification

$counter\_val(c, k) \triangleq c \mapsto k.$

$$\left\{counter\_val(c, k) * k + 1 < 2^b\right\}$$
$$incr\ c$$
$$\{counter\_val(c, k) \vee counter\_val(c, k + 1)\}$$

Weak, worst-case analysis: after 2^b increments cannot call again.
Lost the $O(\log \log n)$ benefit.

Spec 2: a coarse specification

Hide the value inside the predicate:

$$counter(c) \triangleq \exists k.\ counter\_val(c, k)$$

Spec 2: a coarse specification

Hide the value inside the predicate:

$$counter(c) \triangleq \exists k.\ counter\_val(c, k)$$

Charge a flat error budget per call:

$$\left\{ counter(c) * \lightning\left(1/2^{2^b-1}\right) \right\}$$
$$incr\ c$$
$$\{counter(c)\}$$

Spec 2: a coarse specification

Hide the value inside the predicate:

$$counter(c) \triangleq \exists k.\ counter\_val(c, k)$$

Charge a flat error budget per call:

$$\left\{ counter(c) * \lightning\left(1/2^{2^b-1}\right) \right\}$$
$$incr\ c$$
$$\{counter(c)\}$$

The budget is the worst-case failure probability: at the boundary state $k = 2^b - 1$, any successful coin overflows.

Spec 2: a coarse specification

Hide the value inside the predicate:

$$counter(c) \triangleq \exists k.\ counter\_val(c, k)$$

Charge a flat error budget per call:

$$\left\{ counter(c) * \lightning\left(1/2^{2^b-1}\right) \right\}$$
$$incr\ c$$
$$\{counter(c)\}$$

The budget is the worst-case failure probability: at the boundary state $k = 2^b - 1$, any successful coin overflows.

Better, but still pessimistic: every call pays the boundary cost, even though early calls almost never overflow.

Spec 3: tight analysis with bounded-call budget

Compute the exact overflow probability under n calls from state k:

$$\begin{aligned} err\_overflow(0, k) &= 0 \\ err\_overflow(n+1, k) &= \tfrac{1}{2^k} \cdot e_{inc} + \left(1 - \tfrac{1}{2^k}\right) \cdot err\_overflow(n, k) \end{aligned}$$

where $e_{inc} = err\_overflow(n, k+1)$ if $k + 1 < 2^b$, else 1.

Spec 3: tight analysis with bounded-call budget

Compute the exact overflow probability under n calls from state k:

$$\begin{aligned} \mathit{err\_overflow}(0, k) &= 0 \\ \mathit{err\_overflow}(n+1, k) &= \tfrac{1}{2^k} \cdot e_{\mathit{inc}} + \left(1 - \tfrac{1}{2^k}\right) \cdot \mathit{err\_overflow}(n, k) \end{aligned}$$

where $e_{\mathit{inc}} = \mathit{err\_overflow}(n, k+1)$ if $k + 1 < 2^b$, else 1.

Bake the remaining budget into the representation predicate:

$$\mathit{counter\_budget}(c, n) \triangleq \exists k.\ \mathit{counter\_val}(c, k) * \lightning(\mathit{err\_overflow}(n, k))$$

Spec 3: tight analysis with bounded-call budget

Compute the exact overflow probability under n calls from state k:

$$\begin{aligned} err\_overflow(0, k) &= 0 \\ err\_overflow(n+1, k) &= \tfrac{1}{2^k} \cdot e_{inc} + \left(1 - \tfrac{1}{2^k}\right) \cdot err\_overflow(n, k) \end{aligned}$$

where $e_{inc} = err\_overflow(n, k+1)$ if $k + 1 < 2^b$, else 1.

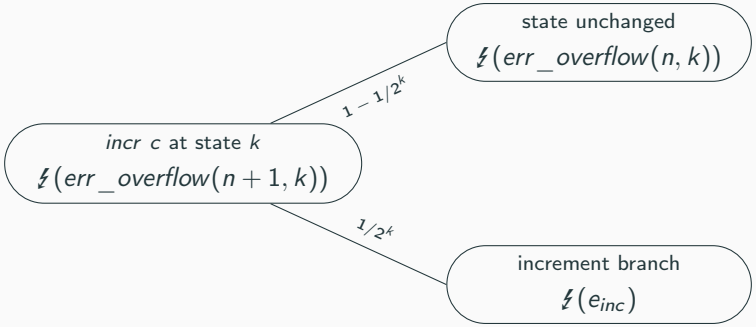
Bake the remaining budget into the representation predicate:

$$counter\_budget(c, n) \triangleq \exists k.\ counter\_val(c, k) * \lightning(err\_overflow(n, k))$$

$$\{counter\_budget(c, n+1)\}\ incr\ c\ \{counter\_budget(c, n)\}$$

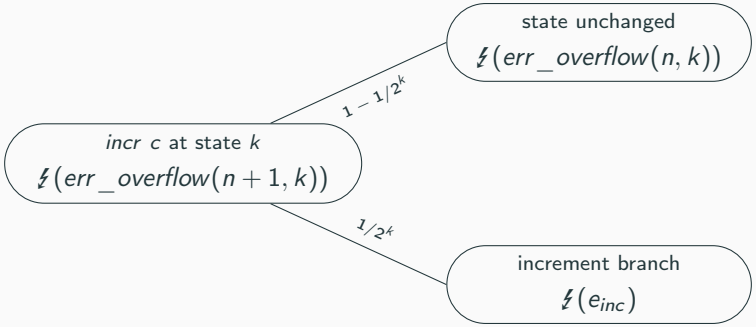
Proof sketch: average across the branches

Initial: state k, budget $\lightning(\mathit{err\_overflow}(n+1,k))$. Split across the outcomes of rand(2^k-1) by the averaging rule:



Proof sketch: average across the branches

Initial: state k, budget $\lightning(\mathit{err\_overflow}(n+1,k))$. Split across the outcomes of $\mathsf{rand}(2^k-1)$ by the averaging rule:



Averaging condition holds by definition of $\mathit{err\_overflow}(n+1,k)$:

$$\tfrac{1}{2^k}\cdot e_{inc} + \left(1-\tfrac{1}{2^k}\right)\cdot \mathit{err\_overflow}(n,k) \;=\; \mathit{err\_overflow}(n+1,k)$$

Proof sketch 2: Reason about the assert

If we increment, then before the assert we have e_{inc} where $e_{inc} = err\_overflow(n, k+1)$ if $k+1 < 2^b$, else 1.

If $k+1 < 2^b$ the assert passes.

Proof sketch 2: Reason about the assert

If we increment, then before the assert we have e_{inc} where $e_{inc} = err\_overflow(n, k+1)$ if $k+1 < 2^b$, else 1.

If $k+1 < 2^b$ the assert passes.

If $k+1 = 2^b$, the assert would fail!

But then $e_{inc} = 1$ and $\lightning(1) \vdash \text{False}$

Almost-Sure Termination with Eris$_t$

Background: Partial vs. Total Correctness

As described so far, Hoare triples have expressed partial correctness:

$$\{P\}\, e\, \{Q\} \quad \equiv \quad \text{if } e \text{ starts in a state satisfying } P \text{ and terminates, the final state satisfies } Q.$$

Trivially satisfied by non-terminating programs.

Background: Partial vs. Total Correctness

As described so far, Hoare triples have expressed partial correctness:

$$\{P\}\, e\, \{Q\} \quad \equiv \quad \text{if } e \text{ starts in a state satisfying } P \text{ and terminates, the final state satisfies } Q.$$

Trivially satisfied by non-terminating programs.

Can also consider total correctness that requires termination:

$$[P]\, e\, [Q] \quad \equiv \quad \text{if } e \text{ starts in a state satisfying } P, \text{ then } e \text{ terminates and the final state satisfies } Q.$$

Total Correctness for Randomized Programs

Eris is similarly a partial logic.

We have developed a total version called Eris$_t$.

In a probabilistic setting, we ask for almost-sure termination: the program terminates with probability 1.

Soundness of Eris$_t$

Total Soundness.
If $[\lightning(\epsilon)]\, e\, [Q]$, then for any starting state, with probability $\geq 1 - \epsilon$, running e will terminate and end in a state satisfying Q.

Soundness of Eris$_t$

Total Soundness.
If $[\lightning(\epsilon)]\, e\, [Q]$, then for any starting state, with probability $\geq 1 - \epsilon$, running e will terminate and end in a state satisfying Q.

Prove almost-sure termination by **continuity**:
If for all $\epsilon > 0$, $[\lightning(\epsilon)]\, e\, [Q]$, then e terminates almost-surely from any starting state.

Soundness of Eris_t

Total Soundness.
If $[\lightning(\epsilon)]\, e\, [Q]$, then for any starting state, with probability $\geq 1 - \epsilon$, running e will terminate and end in a state satisfying Q.

Prove almost-sure termination by **continuity**:
If for all $\epsilon > 0$, $[\lightning(\epsilon)]\, e\, [Q]$, then e terminates almost-surely from any starting state.

Internalize this continuity as a rule:

$$\frac{\forall \epsilon > 0.\ [P * \lightning(\epsilon)]\, e\, [Q]}{[P]\, e\, [Q]}$$

How do the rules change?

The partial logic supports the Löb induction rule we saw before.

Total version removes Löb induction. Must reason about recursion by induction on a well-founded relation.

Otherwise all the same rules.

The Trouble with Induction for Almost-Sure Termination

$$
\begin{aligned}
&\mathsf{let\ rec}\ \mathit{coinToss}\ () = \\
&\quad \mathsf{if}\ \mathsf{rand}(1) = 0\ \mathsf{then}\ () \\
&\quad \mathsf{else}\ \mathit{coinToss}\ ()
\end{aligned}
$$

This terminates with probability 1.

The Trouble with Induction for Almost-Sure Termination

$$\begin{aligned}&\mathsf{let\ rec}\ \mathit{coinToss}\ () = \\ &\quad \mathsf{if}\ \mathsf{rand}(1) = 0\ \mathsf{then}\ () \\ &\quad \mathsf{else}\ \mathit{coinToss}\ ()\end{aligned}$$

This terminates with probability 1.

But what is "getting smaller" on each recursive call?

Key Idea: Error Credit Induction

$$\begin{array}{l}\mathsf{let\ rec}\ \mathit{coinToss}\ () = \\ \quad \mathsf{if}\ \mathsf{rand}(1) = 0\ \mathsf{then}\ () \\ \quad \mathsf{else}\ \mathit{coinToss}\ ()\end{array}$$

Let's suppose we start with $↯(\epsilon)$ for some $\epsilon > 0$.

Recall: $↯(1) \vdash \mathsf{False}$

Key Idea: Error Credit Induction

$$\begin{array}{l}\mathsf{let\ rec}\ \mathit{coinToss}\ () = \\ \quad \mathsf{if}\ \mathsf{rand}(1) = 0\ \mathsf{then}\ () \\ \quad \mathsf{else}\ \mathit{coinToss}\ ()\end{array}$$

Let's suppose we start with $↯(\epsilon)$ for some $\epsilon > 0$.

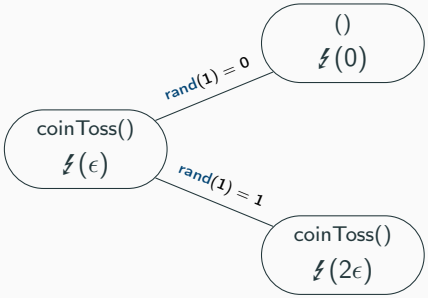
Recall: $↯(1) \vdash \mathsf{False}$

Each iteration we can **double** the credits we have for recursive call.

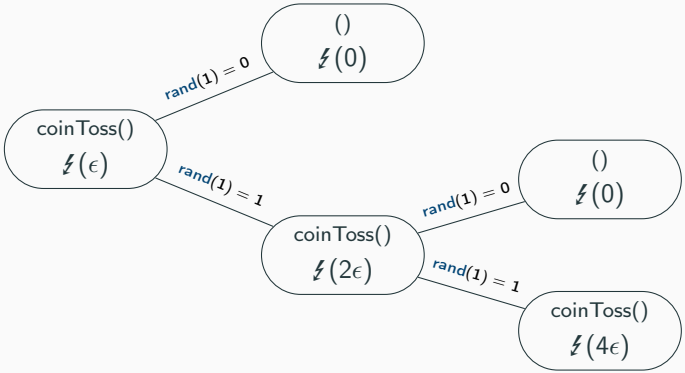
Induct on number of rounds until getting $↯(1)$.

Error Credit Induction for coinToss

Error Credit Induction for coinToss



Error Credit Induction for coinToss



No matter how small ϵ, eventually get $\lightning(1)$.

Error Credit Induction Rule

This induction principle is abstractly captured by:

$$\frac{\epsilon > 0 \qquad k > 1 \qquad \forall \epsilon'.\ (\lightning(k \cdot \epsilon') \mathrel{-\!\!*} P) * \lightning(\epsilon') \vdash P}{\lightning(\epsilon) \vdash P}$$

Error Credit Induction Rule

This induction principle is abstractly captured by:

$$\frac{\epsilon > 0 \qquad k > 1 \qquad \forall \epsilon'.\ (\lightning(k \cdot \epsilon') \mathrel{-\!\!*} P) * \lightning(\epsilon') \vdash P}{\lightning(\epsilon) \vdash P}$$

To use: own $\lightning(\epsilon)$ and pick amplification factor $k > 1$.

Get an arbitrary $\lightning(\epsilon')$ and an IH usable after paying $\lightning(k \cdot \epsilon')$.

Error Credit Induction Rule

This induction principle is abstractly captured by:

$$\frac{\epsilon > 0 \qquad k > 1 \qquad \forall \epsilon'.\ (\lightning(k \cdot \epsilon') \mathrel{-\!\!*} P) * \lightning(\epsilon') \vdash P}{\lightning(\epsilon) \vdash P}$$

To use: own $\lightning(\epsilon)$ and pick amplification factor $k > 1$.

Get an arbitrary $\lightning(\epsilon')$ and an IH usable after paying $\lightning(k \cdot \epsilon')$.

Soundness: induct on $\lceil -\log_k(\epsilon) \rceil$, the number of scalings until reaching $\lightning(1)$.

Completeness

Eris$_t$ is relatively **complete** for allocation-free programs.

Meaning: if e is an expression that:

1. almost-surely terminates
2. does not get stuck
3. does not allocate

then you can derive a triple in Eris$_t$ for e.

Distribution Correctness

Only bounding error/termination might seem **limited**. But can characterize arbitrary distributions this way.

Distribution Correctness

Only bounding error/termination might seem **limited**. But can characterize arbitrary distributions this way.

Suppose f is an implementation of a sampler for a distribution D and we want to prove f is correct.

Marionneau et al. extend Eris_t soundness proof to show that it suffices to establish

$$\frac{\mathbb{E}_{x \sim D}[\epsilon(x)] = \epsilon_0}{[\lightning(\epsilon_0)]\, f\ ()\, [x.\ \lightning(\epsilon(x))]}$$

Case Studies for Eris$_t$

Termination:

- Higher-order rejection sampler
- Incremental randomized SAT solving
- 1-D symmetric random walk

Sampler Correctness:

- Geometric
- Binomial
- Negative Binomial
- Beta Binomial
- ...

Rocq Exercises