Alerus

Verifying Probabilistic Programs in Rust

Alexander Bai, NYU
Joseph Tassarotti, NYU

Randomness is an Essential Tool

- Cryptography: randomness of keys, hashes, pseudo-randomness functions…
- Differential Privacy: randomized response, Laplace & Gaussian Mechanism…
- Data Structure and Algorithms: Bloom Filters, Miller-Rabin, Skip Lists…
- Machine Learning: Stochastic Gradient Descent, Markov Chain Monte Carlo...

Probabilistic Verification

- Pre-Expectation Calculus: Caesar / HeyVL
  - Assertions: $\mathrm{State} \to \mathbb{R}$
- Relational Hoare Logic: EasyCrypt
  - Assertions: $\mathrm{State} \to \mathrm{State} \to \mathrm{Prop}$
- Probabilistic Separation Logic: PSL, LiLac, Bluebell
  - Assertions: $\mathrm{Distr}(\mathrm{State}) \to \mathrm{Prop}$

- Their interpretations of assertions are different from ordinary verifiers
- Only target toy languages, missing essential features (pointers, closures, higher order state, etc).

Motivation

- We want to add support for probability in a lightweight manner in Verus
    - Not a rewrite of a Verus
    - No alien verification condition encoding
    - Usual program logic rules are preserved

This work: Alerus

- **Lightweight Extension** of Verus that supports reasoning about probabilistic programs, using axiomatized ghost state
- Supports verifying Almost Sure Termination with decrease clause
- Verified Samplers:
  - Discrete Laplace/Gaussian Sampler from OpenDP
  - Alias Method
  - Fast Loaded Dice Roller
- Soundness by adapting Verusbelt

Bigger Picture

- Inspiration: Eris [ICFP'24], adds probabilistic reasoning to Iris through a ghost state called error credits

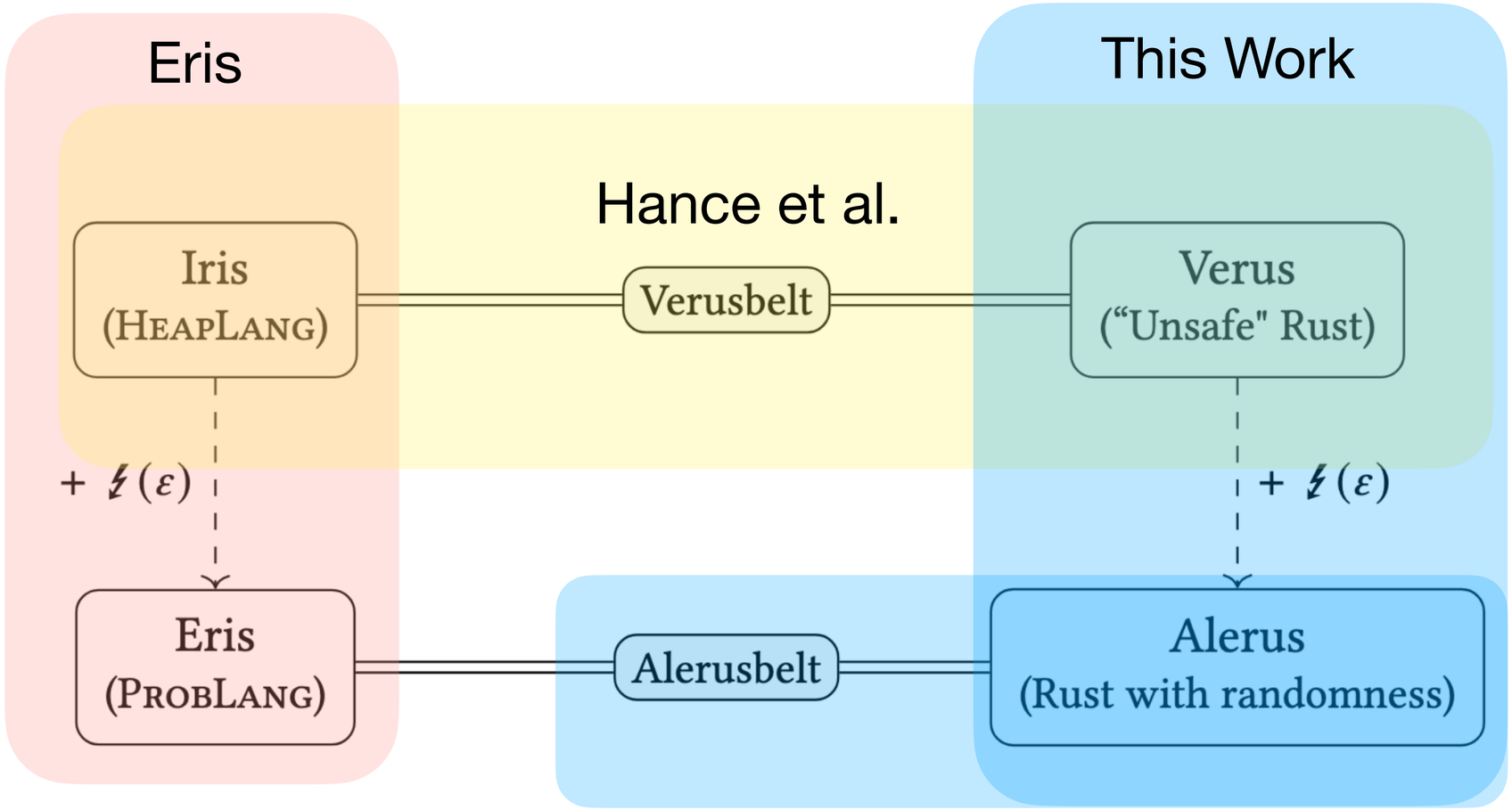


Fig. 1. High Level Picture of Alerus

Bigger Picture

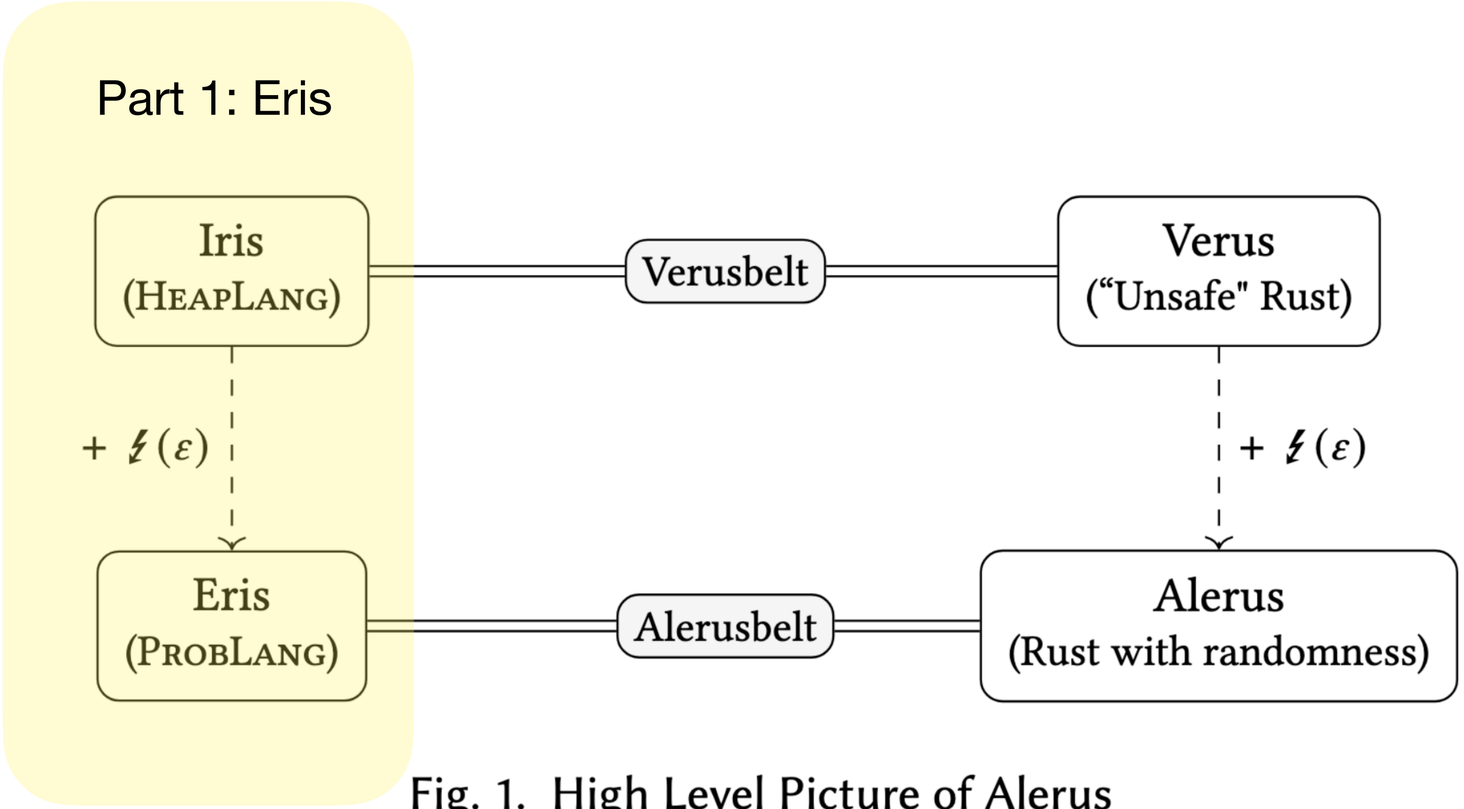


Fig. 1. High Level Picture of Alerus

Eris: Resourceful Reasoning about Error Probabilities

$$\begin{aligned}&\text{let } a = \text{rand } 2^{64} \text{ in}\\&\text{let } b = \text{rand } 2^{64} \text{ in}\\&\quad \text{assert}(a \neq b)\end{aligned}$$

Motivation: Hash Collision

This assertion fails with probability $\frac{1}{2^{64}}$

$\text{rand } 2^{64}$

samples uniformly randomly from $\{0, \ldots, 2^{64} - 1\}$

Eris: Resourceful Reasoning about Error Probabilities

$$\boxed{\{P * \lightning(\varepsilon)\}\ e\ \{Q\}}$$

$\lightning(\varepsilon)$ means owning ε error credits

If P holds and we own ε error credits,
then e terminates and postcondition Q holds
with probability at least $1 - \varepsilon$

Eris: Resourceful Reasoning about Error Probabilities

$$\{ \lightning\,(1/2^{64}) \}$$

$$\text{let } a = \text{rand } 2^{64} \text{ in}$$

$$\text{let } b = \text{rand } 2^{64} \text{ in}$$

$$(a, b)$$

$$\{ (a, b).\ a \neq b \}$$

The postcondition will only fail with probability $\frac{1}{2^{64}}$

Error Credits

Derived Rule

$$\forall x. \quad \vdash [\lightning (1/n)]\ \mathsf{rand}\, n\, [v.\ v \neq x]$$

Error Credits

$\{ \lightning (1/2^{64}) \}\ \mathsf{rand}\ 2^{64}\ \{ v.\ v \neq a \}$

Example

$\{ \lightning (1/2^{64}) \}$

$\quad \mathsf{let}\ a = \mathsf{rand}\ 2^{64}\ \mathsf{in}$

$\quad \mathsf{let}\ b = \mathsf{rand}\ 2^{64}\ \mathsf{in}$

$\quad\quad (a, b)$

$\{ (a, b).\ a \neq b \}$

Error Credits

$\{ \lightning (1/2^{64}) \}\ \mathsf{rand}\ 2^{64}\ \{ v.\ v \neq a \}$

Derived Laws

$\{ \lightning (1/2^{64}) \}$

$\quad \mathsf{let}\ a = \mathsf{rand}\ 2^{64}\ \mathsf{in}$

$\{ \lightning (1/2^{64}) \}$

$\quad \mathsf{let}\ b = \mathsf{rand}\ 2^{64}\ \mathsf{in}$

$\{ b \neq a \}$

$\quad \mathsf{assert}(a \neq b)$

Error Credits

Primitive Rules

Explosion $\lightning(1) \vdash \bot$

Error Credits

Primitive Rules

Explosion $\lightning(1) \vdash \bot$

Splitting $\lightning(\varepsilon_1 + \varepsilon_2) \dashv\vdash \lightning(\varepsilon_1) \ * \ \lightning(\varepsilon_2)$

Error Credits

Primitive Rules

Explosion $\lightning(1) \vdash \bot$

Splitting $\lightning(\varepsilon_1 + \varepsilon_2) \dashv\vdash \lightning(\varepsilon_1) * \lightning(\varepsilon_2)$

Expectation Preservation

$$\frac{\forall i < N.\ 0 \leq F(i) \leq 1 \quad \frac{1}{N}\sum_{i=0}^{N-1} F(i) = \varepsilon}{\vdash [\lightning(\varepsilon)]\ \mathsf{rand}\ N\ [v.\ \lightning(F(v))]}$$

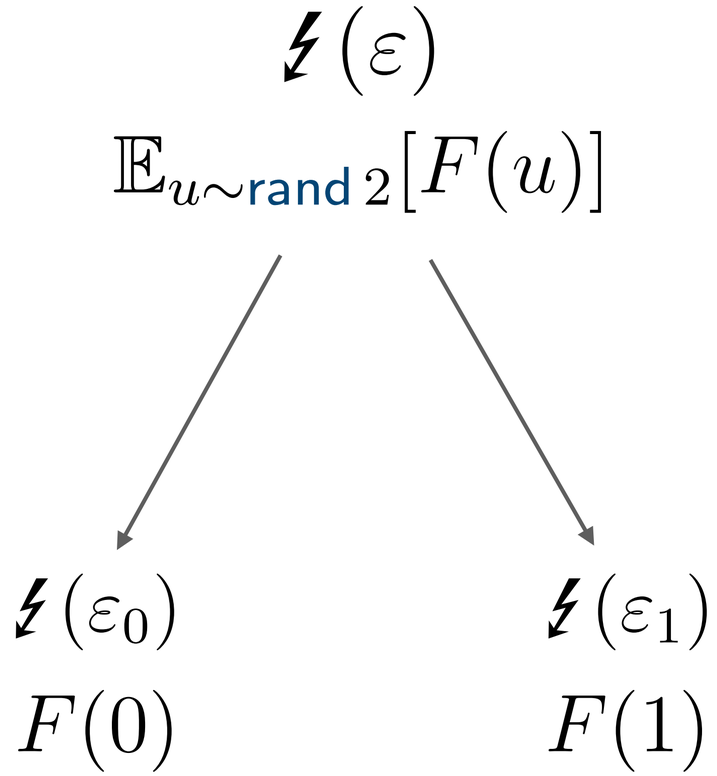
The key work horse for proving programs with rand

Error Credits

Primitive Rules

Explosion $\quad \lightning(1) \vdash \bot$

Splitting $\quad \lightning(\varepsilon_1 + \varepsilon_2) \dashv\vdash \lightning(\varepsilon_1) * \lightning(\varepsilon_2)$



Expectation Preservation

$$\frac{\forall i < N.\ 0 \le F(i) \le 1 \quad \frac{1}{N}\sum_{i=0}^{N-1} F(i) = \varepsilon}{\vdash [\lightning(\varepsilon)]\ \text{rand}\ N\ [v.\ \lightning(F(v))]}$$

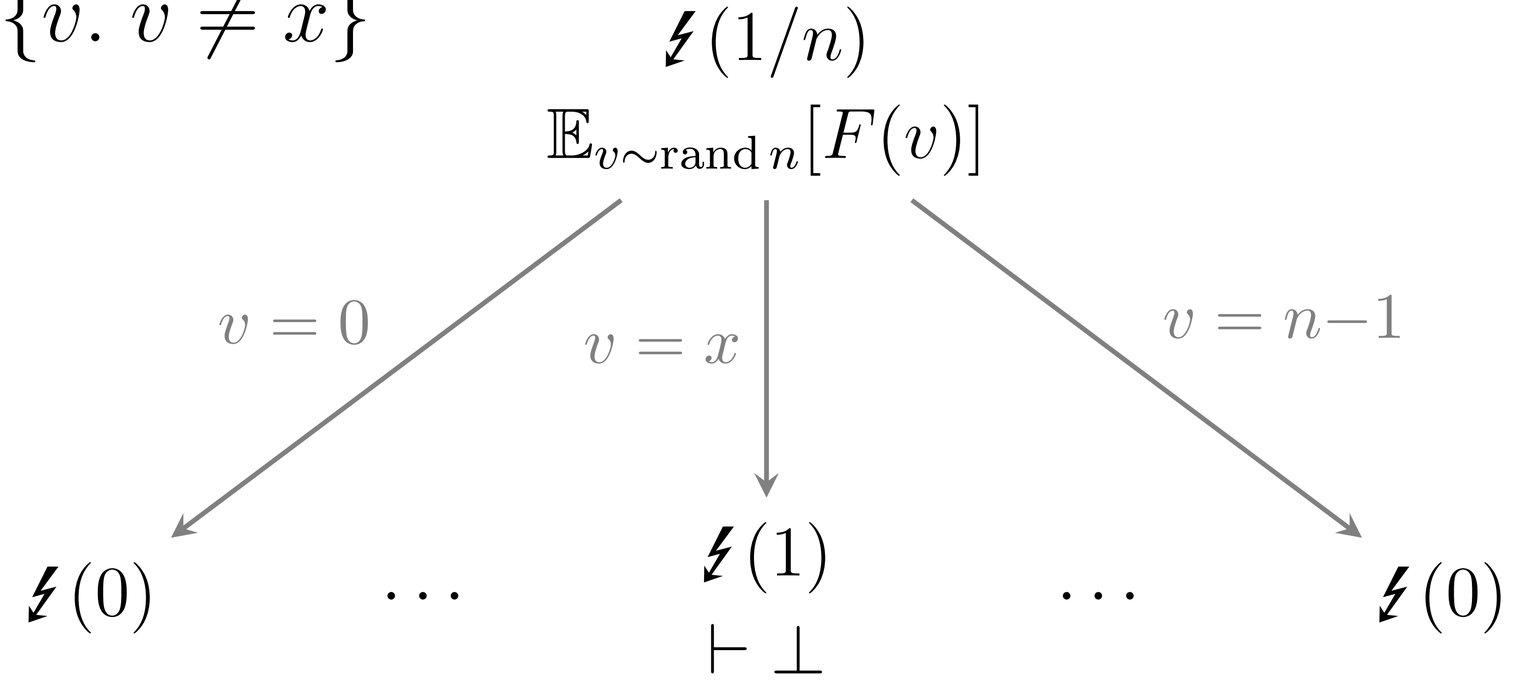
Expectation Preservation

Prove the derived rule

$$\frac{\forall i < N.\ 0 \le F(i) \le 1 \quad \frac{1}{N}\sum_{i=0}^{N-1} F(i) = \varepsilon}{\vdash [\lightning(\varepsilon)]\ \mathsf{rand}\ N\ [v.\ \lightning(F(v))]}$$

$$\forall x. \quad \vdash \{\lightning(1/n)\}\ \mathsf{rand}\, n\, \{v.\ v \neq x\}$$

$$F(v) = \begin{cases} 1 & v = x \\ 0 & v \neq x \end{cases}$$



Bigger Picture

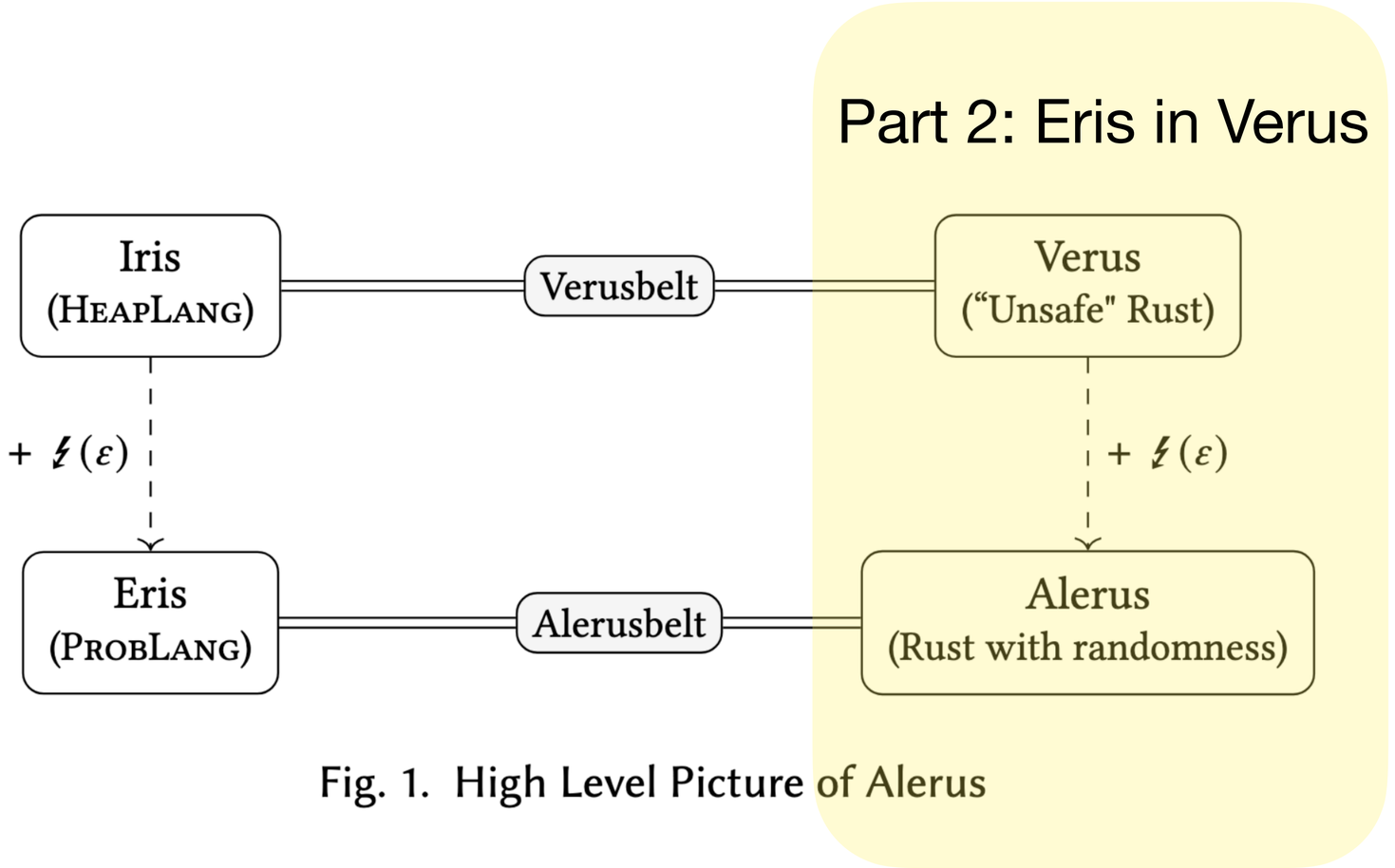


Fig. 1. High Level Picture of Alerus

Encoding Eris in Verus

- Define the Error Credit Resource Algebra by instantiating the PCM and Resource trait
- Prove the algebraic rules (splitting & explode) in Verus
- Axiomatize primitive rule of how rand changes the error credit resource (expectation preservation)-
- About 230 LoC, no changes to Verus codebase

Eris in Verus

```
#[verus::trusted]
#[verifier::external_body]
pub fn rand_ubig(
    bound: &UBig,
    Tracked(e): Tracked<ErrorCreditResource>,
    Ghost(f): Ghost<spec_fn(real) -> real>,
) -> ((n, out_credit): (UBig, Tracked<ErrorCreditResource>))
    requires
        ubig_view(bound) > 0,
        forall |i: nat| (#[trigger] f(i as real)) >= 0real,
        exists |eps: real| (ErrorCreditCarrier::Value { car: eps } =~= e.view())
            && eps >= average_nat(ubig_view(bound), f),
    ensures
        ubig_view(&n) < ubig_view(bound),
        (ErrorCreditCarrier::Value { car: f(ubig_view(&n) as real) }) =~= out_credit.view().view(),
{
    let val = random::rand_ubig(bound.clone());
    (val, Tracked::assume_new())
}
```

Expectation Preservation

$$\frac{\forall i < N.\ 0 \le F(i) \le 1 \qquad \frac{1}{N}\sum_{i=0}^{N-1} F(i) = \varepsilon}{\vdash [\lightning(\varepsilon)]\ \mathsf{rand}\ N\ [v.\ \lightning(F(v))]}$$

Almost Sure Termination in Verus

Almost Sure Termination

```
fn geometric() -> u64 {
    if flip() {           // heads (prob 1/2): stop
        0
    } else {              // tails (prob 1/2): one more failure, then recurse
        1 + geometric()
    }
}
```

- e almost-surely terminates if it terminates with probability 1
- For geometric sampler, the probability it doesn't terminate is $\lim_{n \to \infty} \frac{1}{2^n} = 0$
- What should we decrease on?
- Lots of program logics have very complicated specialized rules

Eris in Verus

One Last Rule

Thin Air [ICFP'25]

$$\frac{\vdash [P * \exists \varepsilon > 0.\, \lightning(\varepsilon)]\, e\, [v.\, Q(v)]}{\vdash [P]\, e\, [v.\, Q(v)]}$$

```
#[verus::trusted]
#[verifier::external_body]
pub fn thin_air() -> Tracked<ErrorCreditResource>
    ensures exists |eps: real| eps > 0 && ret.view() == Value { car: eps },
{
    Tracked::assume_new()
}
```

Almost Sure Termination (for free)

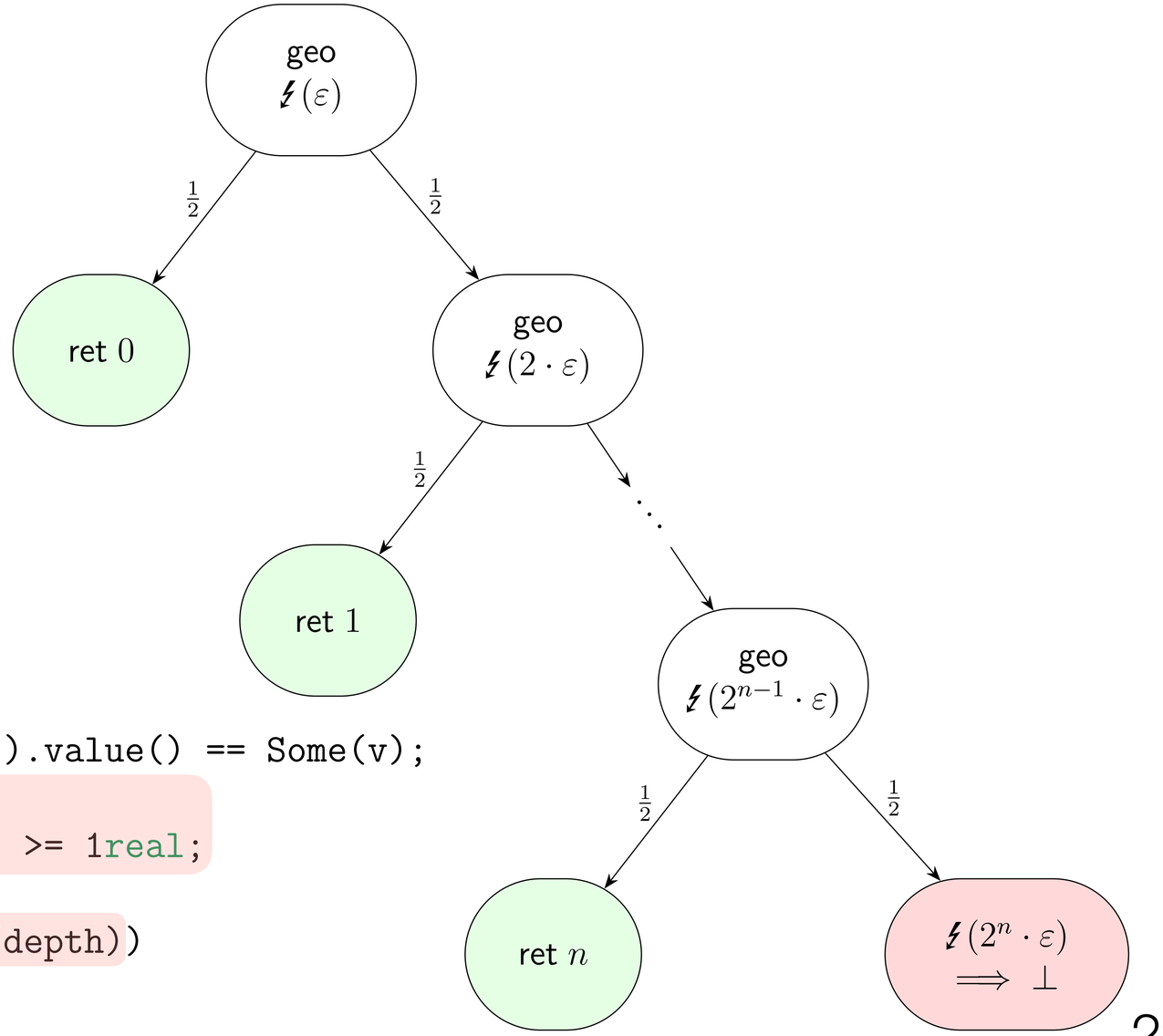
Each recursion:

Invariant: $2^{\text{depth}} \cdot \varepsilon = 1$

$\varepsilon \mapsto 2 \cdot \varepsilon$

$2^{\text{depth}} \cdot \varepsilon \mapsto 2^{\text{depth}-1} \cdot (2 \cdot \varepsilon)$

```
pub fn geometric() -> (ret: u64) {
    let Tracked(input_credit) = thin_air();
    let ghost depth: nat;
    let ghost eps: real;
    proof {
        eps   = choose |v: real| input_credit.view().value() == Some(v);
        archimedean_exp_growth(eps, 2real);
        depth = choose |k: nat| eps * pow(2real, k) >= 1real;
    }
    bounded_geometric(Tracked(input_credit), Ghost(depth))
}
```



Almost Sure Termination (for free)

- More Examples:
  - 1-dimensional random walk
  - Higher Order Rejection Sampler [ICFP'24]
  - Escaping Spline [POPL'18] [ICFP'24]
- If people come up with all these specialized rules for almost-sure termination, are the Eris rules really enough?
- Completeness of Eris [ICFP'26]
  If e almost-surely terminates, we can prove a Eris triple for e

Verify Sampler Correctness

Distributional Specification

- It looks like you can only use Eris to specify particular outcomes
- [CPP'26] presents a general specification that allows you to characterize the exact distribution a program samples from
- You can use this to specify a program samples from the correct distribution!

Expectation Preservation

$$\frac{\forall i < N.\ 0 \leq F(i) \leq 1 \quad \frac{1}{N}\sum_{i=0}^{N-1} F(i) = \varepsilon}{\vdash [\lightning(\varepsilon)]\ \mathsf{rand}\ N\ [v.\ \lightning(F(v))]}$$

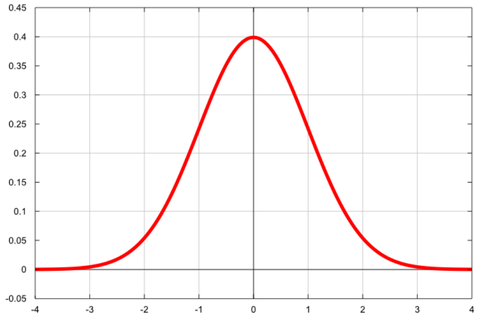
Distributional

$$\frac{\forall v.\ 0 \leq F(v) \leq 1 \qquad \mathbb{E}_{v\sim\mu}[F(v)] \ = \ \varepsilon}{\vdash [\lightning(\varepsilon)]\ e\ [v.\ \lightning(F(v))]}$$

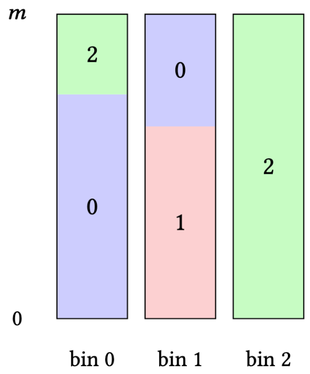
e is sampling from μ

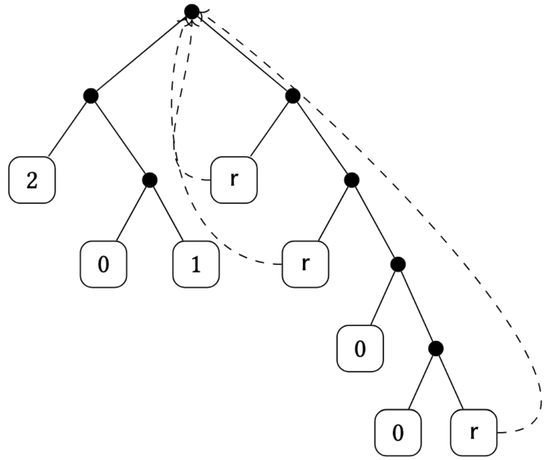
$$\Pr_{\mathrm{exec}\ e}[\lambda w.\ w = v] \ = \ \mu(v)$$

Verified Samplers



- Discrete Laplace/Gaussian Sampler [NeurIPS'20] [PLDI'24]
  - Axiomatized e^x (uninterpreted, 1 nontrivial axiom about Taylor bound)
- Alias Method
- Fast Loaded Dice Roller [AISTATS'20] [FM'26]
  - Deterministic preprocessing step, then sampling via random walk





c	$h[c]$	$lab[c]$
1	0	[ ]
2	2	[2, r]
3	3	[0, 1, r]
4	1	[0]
5	2	[0, r]

Verusbelt → Alerusbelt

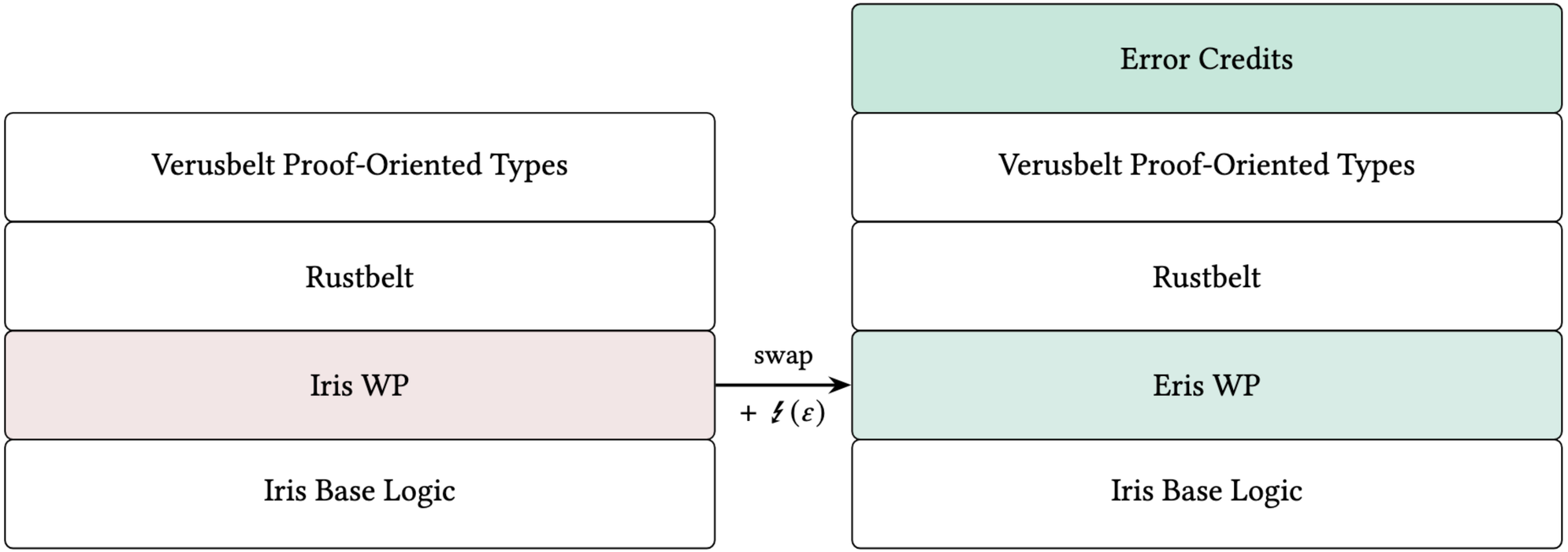


Fig. 4. The Verusbelt Proof-Oriented type proofs are written against a generic Iris weakest precondition, so porting Verusbelt to a probabilistic model is a *swap* of the underlying semantic model—from Iris's `wp_pre` to Eris's distributional `pgl_wp_pre`, additionally threading error credits $↯(\varepsilon)$ through the WP. The shared separation-logic rules and the Iris base logic underneath let the proof layer carry over.

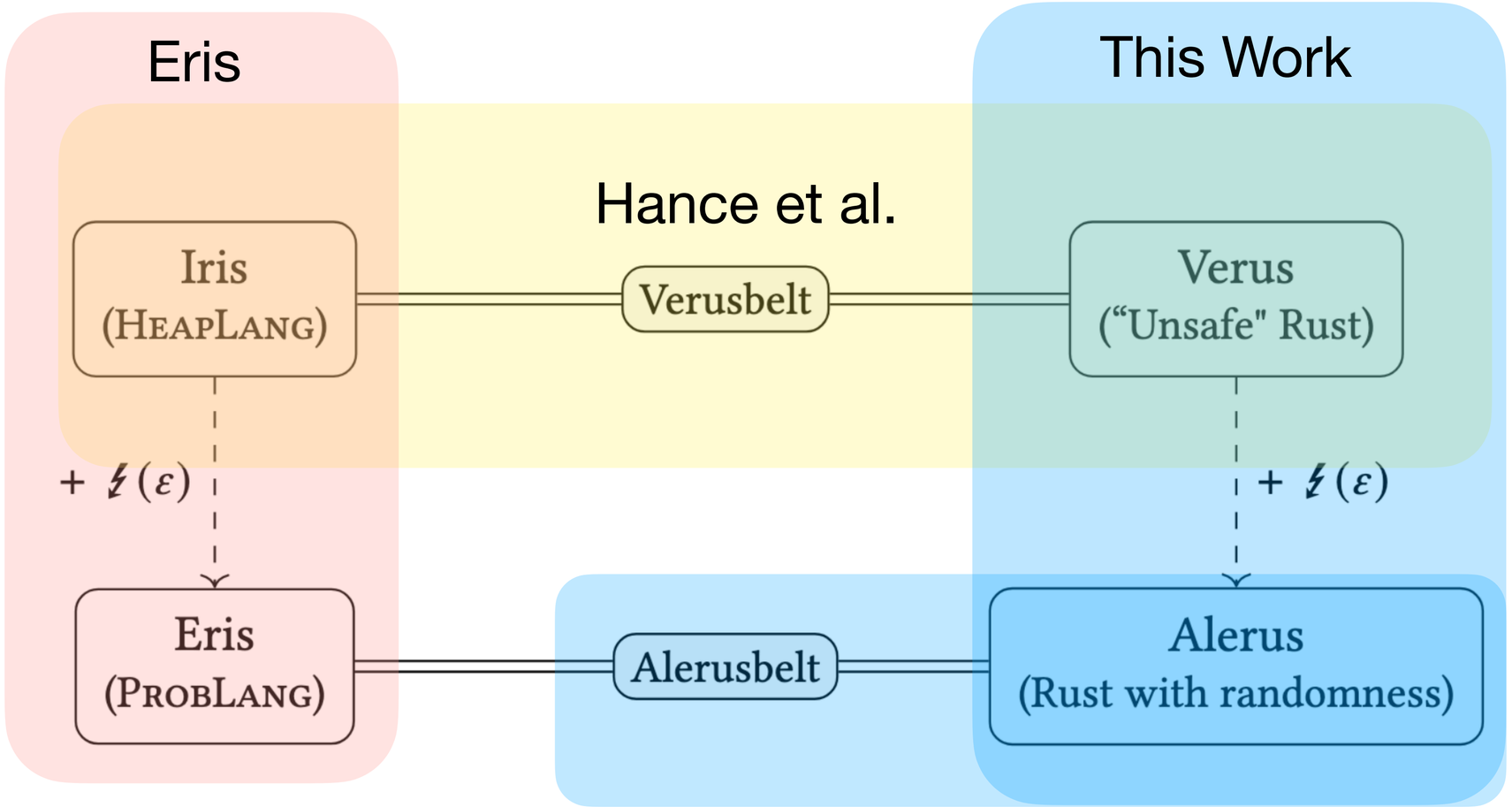


Fig. 1. High Level Picture of Alerus

References

- Clutch: clutch-project.org
- Verusdoc: https://ahuoguo.github.io/alerus/
- Github: https://github.com/ahuoguo/alerus/