Verifying Probabilistic Programs in Rust

ALEXANDER Y. BAI, New York University, USA
JOSEPH TASSAROTTI*, New York University, USA

Recent work has developed many techniques for formally verifying probabilistic programs. However, existing verification frameworks for probabilistic programs are restricted to custom, idealized languages designed for verification. As a result, they cannot be used to verify off-the-shelf probabilistic programs written in standard languages. In contrast, for non-probabilistic programs, a number of verification tools now support verifying realistic code written in widely used languages such as Go, C, and Rust. To verify probabilistic programs written in these languages, it would be useful to be able to reuse, as much as possible, the extensive development work that has gone into such tools.

This paper presents Alerus, a framework for verifying probabilistic Rust programs. Alerus is based on Verus, a verification tool for Rust that supports SMT-based automation and separation-logic-inspired reasoning features. Alerus extends Verus with support for probabilistic reasoning while retaining these expressive features. To do so, Alerus uses a lightweight encoding of probabilistic *error credits*, a form of ghost state for randomized reasoning introduced in the Eris program logic. By deriving an appropriate specification using error credits, Alerus supports verifying the correctness of randomized sampling algorithms. We use this technique to verify several sampling routines for discrete distributions, including samplers for the discrete Laplace and discrete Gaussian distributions, the alias method, and the fast loaded dice roller.

We establish the soundness of our error credit extension by adapting VerusBelt, a recently developed logical relations model of Verus that encodes its features in terms of the Iris separation logic. To do so, we replace the use of Iris's standard weakest precondition in this model with Eris's probabilistic weakest precondition instead. The resulting soundness proof is fully mechanized in Rocq.

Additional Key Words and Phrases: Formal Verification, Probabilistic Programs, Rust

1 Introduction

Probabilistic programs are challenging to implement correctly, and their randomized behavior can make them difficult to test. At the same time, bugs in probabilistic programs can have critical consequences, especially in security applications, where randomness is an essential part of algorithms for cryptography and differential privacy. Because of these important applications, there has long been interest in developing program logics to formally verify the correctness of probabilistic programs. As a result, prior work has developed a wide range of probabilistic program logics [5–8, 15, 35]. However, existing probabilistic program logics cannot be used to verify probabilistic programs written in standard, full-fledged programming languages, because at present, these logics only support special, idealized languages. While these languages are suitable for modeling core aspects of important randomized programs and data structures, there is a gap between these models and real implementations.

Meanwhile, for *non-probabilistic* programs, there is now a range of program-logic-based tools and frameworks for verifying complex programs written in languages such as Go, C, and Rust [2, 10, 26, 27, 48]. These tools often come with sophisticated features that are needed to reason about the challenging patterns found in real-world programs, and have been used to verify substantial systems written in these languages.

In light of these successes, a natural question arises as to whether these non-probabilistic program logics can be extended to support probabilistic reasoning, or whether probabilistic program logics can be extended to support full-featured languages. Unfortunately, both of these routes are

*Also affiliated with Amazon Web Services. This paper does not reflect the views of Amazon Web Services.

Authors' Contact Information: Alexander Y. Bai, New York University, USA, ayb5065@nyu.edu; Joseph Tassarotti, New York University, USA, jt4767@nyu.edu.

challenging to carry out with many of the approaches that are commonly used to construct probabilistic program logics. The core issue is that many probabilistic logics are structured in a way that is radically different from how standard non-probabilistic verification tools work. For example, logics based on weakest preexpectation transformers [35] change program logic assertions so that they are no longer predicates on program states, but are instead functions from program states to real numbers. In another direction, other probabilistic program logics such as Ellora [5], PSL [8], and Lilac [29] make assertions into predicates over *distributions* of program states. Reconciling these foundations with the approaches that non-probabilistic verification tools use for automation and support for reasoning about heaps and pointers is an open research problem. Another issue is that many modern non-probabilistic program logics make use of rich forms of *ghost state*. Despite promising recent efforts to develop analogous theories of ghost state for probabilistic program logics [31], the current state of the art for probabilistic ghost state still does not have the full flexibility found in non-probabilistic verification.

However, there is one class of program logics for probabilistic verification that appears to be easier to reconcile with state-of-the-art non-probabilistic program verification techniques. These *lifting-based* logics do *not* change the type of assertions. Instead, some limited aspect of a program's probabilistic behavior is tracked through a mechanism like ghost state. Then, just the weakest precondition or Hoare triple of the logic is altered to give this ghost state a probabilistic interpretation. This approach was pioneered by pRHL [7], where it was used for relational probabilistic reasoning. Subsequent lines of work have applied the lifting-based approach in a number of program logics [6, 15, 19, 20] for both unary and relational properties. Some of these logics have recently been based on the Iris separation logic framework, showing that the lifting-based approach can combine the expressive separation logic features of Iris with probabilistic reasoning. Still, that prior work has focused on a toy, idealized Mini-ML-like language, and cannot be used to reason about actual executable programs written in a realistic language.

This paper presents Alerus, a verification framework for reasoning about probabilistic programs written in Rust. Alerus is based on Verus [26, 27], a widely used SMT-based semi-automated verification tool for Rust. Alerus uses a lifting-based approach to incorporate probabilistic reasoning into Verus by extending Verus with *probabilistic error credits*, a form of ghost state for tracking probabilities of events that was introduced in the prior Eris program logic [1]. With error credits, program specifications written in Alerus can be used to bound the probability that a program's execution fails to satisfy some property. By proving a specification of an appropriate form, one can use error credits to show that an implementation of a routine for sampling from some probability distribution correctly generates samples with the right probabilities.

Alerus provides support for proving that a probabilistic program terminates *almost surely*, *i.e.*, terminates with probability 1. This form of termination reasoning is known to be challenging. While Verus provides built-in support for proving termination by annotating recursive functions and loops with some well-founded decreasing measure, many almost-surely terminating probabilistic programs have no such obvious decreasing measure. Prior work on program logics and deductive verifiers for almost-sure termination has come up with alternate, subtle proof rules for working around this issue [36, 41], but adapting Verus to support these alternate rules would require challenging engineering effort. We are able to sidestep this issue entirely: Aguirre et al. [1] show an alternate method of establishing almost-sure termination that uses error credits themselves as the object to induct on, and Alerus is able to use an analogous technique with Verus's existing `decreases` clauses.

Because Verus supports rich forms of ghost state, the embedding of error credits in Alerus is relatively lightweight, and only requires adding two axioms to Verus. The first is a specification for a library method for generating random integers, which connects the resulting random samples to

the error credits. The second is a direct translation of one of the primitive rules for error credits found in Eris. On top of this, reasoning about probabilities of events uses Verus's recent support for Z3's real number theory, which we augment with a small library of results about discrete sums.

Although this axiomatic extension is relatively small, one might still wonder whether it is sound. Therefore, to justify the soundness of Alerus's error credit encoding, we build on the recent VerusBelt project [17], which constructs a semantic model of a large subset of Verus through an encoding into the Iris separation logic. At a high level, the VerusBelt model extends the earlier RustBelt [23] model of Rust's type system by incorporating Verus's specification extensions to Rust's types. The model establishes that if a Rust program (written in a core subset) is well-typed using Verus's extensions, then a corresponding Iris weakest precondition assertion holds. Thus, the soundness of Verus (or at least the modeled subset) follows from the soundness of Iris. We adapt this model to include Alerus's types for error credits and the proof rules they support. To do so, we replace the use of "standard" Iris weakest precondition in the VerusBelt model with the Eris program logic's weakest preconditions instead. This requires generalizing Eris in several ways, including extending it to cover the core Rust-like language used in VerusBelt, as well as incorporating certain Iris features that were missing from the original Eris. Just as the original VerusBelt shows that whenever a program has a certain Verus specification, a corresponding Iris weakest precondition must hold, our adaptation shows that whenever a program satisfies a specification using Alerus's extensions, a corresponding Eris weakest precondition holds. The soundness theorem of Eris then transfers to such programs. This semantic model is fully mechanized in Rocq.

Finally, we demonstrate Alerus by using it to verify a number of sampling algorithms for discrete distributions written in Rust. These include the OpenDP [45] differential privacy library's implementation of samplers for the discrete Laplace and discrete Gaussian, as well as the Fast Loaded Dice Roller (FLDR) [39] and an efficient implementation of Walker's Alias Method [42, 46, 47]. These first two examples demonstrate Alerus's ability to verify realistic samplers from security-critical applications, while the latter two involve stateful preprocessing, which works well with Verus's existing support for reasoning about state manipulation. In addition to using Verus's Z3 automation, we employ Claude Opus 4.7 and 4.8, which are effective at constructing Verus proofs and can handle the mathematical reasoning about probabilities in these examples.

Contributions. To summarize, the contributions of this paper are:

- Alerus, the first verification framework for probabilistic programs that supports verifying off-the-shelf programs written in a realistic, modern programming language.
- A mechanized soundness proof for a core subset of Alerus, based on an adaptation of the VerusBelt semantic model.
- A substantial set of examples of sampling algorithms verified using Alerus.

2 Background

This section gives a brief overview of Verus and then explains how Eris's error credits work.

2.1 Verus

Verus is a semi-automated verification tool for Rust programs [26, 27]. Like Dafny [28], it generates verification conditions that are discharged by an SMT solver. The developer annotates functions with `requires`/`ensures` contracts and loops with `invariants`, and Verus solves the resulting proof obligations automatically with Z3 [14]. Code is partitioned into three modes: `exec` code that compiles and runs, `spec` code that provides pure mathematical definitions for use in specifications, and `proof` code for writing lemmas and helping the SMT solver find proofs. The `spec`- and `proof`-mode code is erased before compilation, so verification imposes no runtime cost. For example, in the

snippet below, the `spec` functions `divides` and `is_prime` give pure mathematical definitions; the `proof` function `even_gt_2_isnt_prime` is a lemma Z3 discharges from those definitions; and the executable `is_prime_impl` is checked by Verus to satisfy the specification given by its preconditions and postconditions.

```
spec fn divides(n: int, k: nat) -> bool { n % (k as int) == 0 }

spec fn is_prime(n: nat) -> bool { forall|k: nat| 2 <= k < n ==> !divides(n as int, k) }

proof fn even_gt_2_isnt_prime(i: nat)
  requires i > 2 && is_even(i as int)
  ensures !is_prime(i) { assert(divides(i as int, 2)); }

fn is_prime_impl(n: u64) -> (result: bool) // only this exec fn will not be erased
  requires n >= 2,
  ensures result == is_prime(n as nat)
{ /* ... implementation and proof ... */ }
```

A distinguishing feature of Verus is how it reasons about *unsafe* Rust. Safe Rust forbids aliased mutable state, but systems code routinely escapes that fragment through raw pointers and interior mutability, despite being semantically safe. Verus reasons about this unsafe code by pairing mutable state with a *permission* token. For example, a permissioned pointer of type `PPtr<T>` cannot be written to directly on its own. Instead, the right to access the underlying data is a separate ghost token `PointsTo<T>`, which carries the location's ghost value. This is the Verus analogue of the separation-logic points-to assertion $\ell \mapsto v$. Reading or writing through the pointer consumes and returns this token in the operation's `requires` and `ensures` clauses. For instance, allocating a heap cell of type `PPtr<u64>` returns the raw pointer together with its `PointsTo` token, and every access threads that token through the operation's contract:

```
let (ptr, Tracked(mut perm)) = PPtr::<u64>::new(7); // ptr |-> 7;  perm is the token
let x = ptr.read(Tracked(&perm));                   // read borrows the token:  x == 7
ptr.write(Tracked(&mut perm), x + 1);               // write mutates it: now ptr |-> 8
let z = ptr.take(Tracked(&mut perm));               // move the value out: ptr |-> uninit;
    z == 8
ptr.free(Tracked(perm));                            // free consumes the token
```

The `PointsTo` token is just one instance of a more general facility for tracking logical permissions associated with state. Specifically, variables in Verus have three modes: `ghost`, `tracked`, and `exec`. The first two are erased at compile time and are used for specifications and proofs. The `ghost` values are duplicable, while `tracked` values, like `PointsTo`, are *affine*. This means they cannot be copied, and the type checker tracks the transfer of their ownership through the program just like standard Rust ownership of physical state. A function consumes the `tracked` arguments it is passed and must return any it intends to give back. This makes `tracked` state behave like a separation-logic resource. While Verus uses Rust's substructural type system to manage these affine resources, the SMT solver checks the validity of specifications [16, 17].

As in modern separation logic frameworks like Iris, Verus allows developers to define custom forms of ghost state that can be used to encode appropriate forms of permissions [16, 17]. This ghost state needs to support various operations, which are required to satisfy certain algebraic laws, like the *resource algebras* used for ghost state in Iris. Concretely, Verus provides a two-layer interface for defining new ghost state: a `ResourceAlgebra` trait with a composition `op` and a validity predicate `valid`, along with the associated algebraic laws, and a `PCM` trait that additionally provides a `unit`. To create a new form of ghost state, the user chooses a carrier type `P`, and implements these traits.

```
trait ResourceAlgebra: Sized {
    spec fn valid(self) -> bool;
    spec fn op(a: Self, b: Self) -> Self;
```

```
    proof fn associative(a: Self, b: Self, c: Self)
        ensures Self::op(a, Self::op(b, c)) == Self::op(Self::op(a, b), c);
    proof fn commutative(a: Self, b: Self)
        ensures Self::op(a, b) == Self::op(b, a);
    proof fn valid_op(a: Self, b: Self)
        requires Self::op(a, b).valid(), ensures a.valid();
}

trait PCM: ResourceAlgebra {
    spec fn unit() -> Self;                    // identity for op
    proof fn op_unit(self) ensures Self::op(self, Self::unit()) == self;
    proof fn unit_valid() ensures Self::unit().valid();
}
```

After establishing the `PCM` trait for a type `P`, the user can then instantiate the `tracked` type `Resource<P>`. These resources live at a ghost location `loc()`, and come equipped with proof functions `alloc` (creates a new piece of ghost state), `join`/`split` (composition and decomposition via `op`), and `validate` (which establishes that the owned value satisfies `valid`). This ghost state can be modified using `update`, which allows the ghost state to be changed in a *frame-preserving* way, meaning that the updated ghost state must be compatible with other possible parts of that ghost state that might be owned elsewhere. Having such a `Resource<P>` in a context is analogous to owning the corresponding separation-logic resource assertion.

Another important feature of Verus is its support for reasoning about termination. This feature is necessary for `spec` and `proof` code to prevent unsoundness from circular reasoning [27]. Verus checks termination by requiring that recursive functions and loops are annotated with a `decreases` clause, which specifies a well-founded measure that decreases on each recursive call or loop iteration. For example, the following recursive `spec` function `factorial` is accepted because its argument `n` is a natural number that strictly decreases on each recursive call, making it a valid well-founded measure:

```
spec fn factorial(n: nat) -> nat
  decreases n,
{
  if n == 0 { 1 } else { n * factorial((n - 1) as nat) }
}
```

Finally, Verus offers flexible ways to attach trusted specifications to executable code. This flexibility is useful for modeling external aspects of the execution environment or library functions. A function marked `#[verifier::external_body]` has a compiled body that Verus treats as opaque (thus "external" to Verus), and its specification is trusted. For example, this is how raw-pointer access itself is given a specification. The `ptr_ref` function dereferences a `*const T`, and its trusted contract uses the `PointsTo` permission to justify the `unsafe` body.

```
#[verifier::external_body]
pub fn ptr_ref<T>(ptr: *const T, Tracked(perm): Tracked<&PointsTo<T>>) -> (v: &T)
    requires
        perm.ptr() == ptr, perm.is_init(),   // must point to initialized memory
    ensures
        v == perm.value(),                   // the borrow reads that value
    opens_invariants none
    no_unwind
{
    unsafe { &*ptr }                         // real unsafe Rust, unchecked by Verus
}
```

The specification we write for `#[verifier::external_body]` extends the trusted computing base (TCB), and the developer takes on the obligation that the implementation actually meets it. Verus

includes axiomatized rules for primitives like `PPtr` and `PCell` in this way. The VerusBelt project [17] subsequently constructed a soundness proof to justify these axioms. As we will see, Alerus uses the same facility to axiomatize aspects of the probabilistic ghost state it uses.

2.2 Eris

As mentioned in the introduction, Eris is a separation logic for proving upper bounds on error probabilities. The language targeted by Eris is an idealized, probabilistic Mini-ML-like language called ProbLang. Roughly speaking, one can think of this language as a sequential version of the default HeapLang that ships with Iris, extended with a command for generating random samples. Aguirre et al. [1] present two different versions of the Eris program logic: one for partial correctness and one for total correctness. For compatibility with Verus's support for proving termination, we will focus on the total-correctness version.

The key idea behind Eris is the introduction of *probabilistic error credits*, a separation logic resource that can be used to track upper bounds on error probabilities. Tracking such probabilities is useful because many probabilistic programs and algorithms have some probability of failure. For example, the Miller-Rabin primality test [37, 38] has some probability of incorrectly declaring that a composite number is prime. Other examples show up in cryptography and differential privacy, where a security claim holds except when a rare event occurs, such as a hash collision or an attacker guessing a randomly generated key.

When reasoning about such algorithms, a key goal is to prove an upper bound on this probability of failure. Eris's error credit assertions, which have the form $↯(\varepsilon)$ for $0 \le \varepsilon \le 1$, represent a logical permission to execute actions that might cause an error with probability at most ε. Using the logic, one proves total Hoare triple specifications of the form $[↯(\varepsilon) \ast P]\ e\ [v.\ Q(v)]$. A triple of this form says that if we execute e in a state initially satisfying P, then with probability at least $1 - \varepsilon$, e will terminate with a value v satisfying $Q(v)$. In other words, the initial "budget" of $↯(\varepsilon)$ error credits in the precondition licenses e to "fail" with probability at most ε, where a failing execution involves either (1) violating safety and getting "stuck"; (2) returning a value v for which $Q(v)$ does not hold; or (3) not terminating.

Error credits are used in the proof rules of the logic to exclude reasoning about certain cases or branches of execution in which an error would occur. For example, the following proof rule reasons about a $\texttt{rand}(N)$ command, which samples an integer uniformly from the set $\{0, \ldots, N-1\}$:[1]

$$\textsc{Err-Rand-Spend}\quad \frac{|S| = M}{\vdash \left[↯\left(\frac{M}{N}\right)\right]\ \texttt{rand}(N)\ [v.\ v \notin S]}$$

This rule allows the user to pick a set of integers S of size M, and in the postcondition we are guaranteed that v is *not* in that set S. That is, we have excluded reasoning about the cases where the returned integer is in the set S. Doing so requires M/N error credits from the precondition, because the probability of drawing an element in S is at most M/N, since each number i in $\{0, \ldots, N-1\}$ is selected with probability $1/N$. Because Eris is a separation logic, applying the rule consumes the error credits from the precondition.

In Eris, this rule is in fact a derived rule. At its core, the logic provides just the following four primitive proof rules for error credits:

[1] In Aguirre et al. [1], this command samples from the set $\{0, \ldots, N\}$ instead of $\{0, \ldots, N-1\}$. We adopt the alternative convention here to match the form we later use in Rust.

$$\textsc{Err-Split}\ \frac{\varepsilon_1 \geq 0 \qquad \varepsilon_2 \geq 0}{\lightning(\varepsilon_1 + \varepsilon_2) \dashv\vdash \lightning(\varepsilon_1) * \lightning(\varepsilon_2)} \qquad \textsc{Err-1}\ \frac{}{\lightning(1) \vdash \mathsf{False}} \qquad \textsc{Err-Thin-Air}\ \frac{\forall \varepsilon > 0.\ \vdash [\lightning(\varepsilon) * P]\ e\ [v.\ \Phi(v)]}{\vdash [P]\ e\ [v.\ \Phi(v)]}$$

$$\textsc{Err-Rand-Exp}\ \frac{\forall i < N.\ 0 \leq \mathcal{E}(i) \leq 1 \qquad \frac{1}{N}\sum_{i=0}^{N-1} \mathcal{E}(i) = \varepsilon}{\vdash [\lightning(\varepsilon)]\ \mathtt{rand}(N)\ [v.\ \lightning(\mathcal{E}(v))]}$$

The Err-Split rule allows for splitting and joining error credits across the separating conjunction. This is useful because it allows us to divide up a budget of error credits and pass ownership of a part of the budget to different components or modules that need them. Err-1 allows us to derive False once we have an error credit of 1. Intuitively, this follows because, if we read $\lightning(\varepsilon)$ as permission to fail to satisfy a specification with probability ε, then when $\varepsilon = 1$, we can fail to satisfy the specification with probability 1, so there is nothing to prove. Reading Err-Thin-Air from the bottom up, the rule allows us to extend the precondition by some additional error credit $\varepsilon > 0$. The quantification here says that this extra credit we produce out of "thin air" may be an arbitrarily small positive number. The soundness of this rule follows from a kind of continuity property of the Hoare triple, where we take the limit as $\varepsilon \to 0$.

Finally, Err-Rand-Exp is what connects error credits to random sampling with $\mathtt{rand}(N)$. When applying this rule, the user picks a function $\mathcal{E} : \{0, \ldots, N-1\} \to [0, 1]$, which maps outcomes of the random sampling to numbers in $[0, 1]$. It then says that if we start with an initial budget of ε credits, then when $\mathtt{rand}(N)$ returns v, we will end up with $\mathcal{E}(v)$ error credits in the postcondition. The premise requires that the *expected value* of $\mathcal{E}$ across the random outcomes is equal to the initial budget ε of error credits we started with. Since $\mathtt{rand}(N)$ returns an integer uniformly from the set $\{0, \ldots, N-1\}$, this expected value is computed by taking the average of $\mathcal{E}$ across the outcomes. In particular, by using rule Err-Rand-Exp, we can derive the earlier Err-Rand-Spend by distributing the credits so that we have one credit on the branches where $v \in S$ and zero credits on the other branches. Then, by applying rule Err-1, we can spend the one error credit on the branches where $v \in S$ to derive False, and thus exclude any further obligations on those branches.

Sampler Correctness. At first, it might appear that Eris only allows for proving relatively limited kinds of specifications, because many properties of interest cannot be expressed merely as upper bounds on error probabilities. However, it turns out that appropriately bounding error probabilities suffices to completely characterize the distribution of values that a program can return. In particular, Marionneau et al. [33] proved a stronger soundness theorem for Eris that allows one to prove that a program draws samples from a given distribution. Their key idea is that Err-Rand-Exp encodes the fact that the $\mathtt{rand}(N)$ command samples uniformly from $\{0, \ldots, N-1\}$ by controlling how credits may be redistributed. Thus, by proving a similar credit redistribution specification about a program e, we can specify what distribution of values e generates. Specifically, to prove that a program e samples from distribution μ, their soundness theorem says that it suffices to prove a specification of the form:

$$\textsc{Expectation Preserving Transformation (EPT)}\ \frac{\forall v \in \mathit{Val}\,.\ 0 \leq \mathcal{E}_2(v) \leq 1 \qquad \sum_{v \in \mathit{Val}} \mu(v) \cdot \mathcal{E}_2(v) = \varepsilon_1}{\vdash [\lightning(\varepsilon_1)]\ e\ [v.\ \lightning(\mathcal{E}_2(v))]}$$

where the sum in the premise is computing the expected value of $\mathcal{E}_2$ under distribution μ, and the rule requires this expected value to be equal to ε_1.

3 Adding Eris-Style Reasoning to Verus

Alerus adds Eris's error credits to Verus without modifying the verifier. Just as Verus uses `tracked` ghost state to implement permissions analogous to separation logic's points-to, Alerus uses tracked ghost state to represent Eris's error credits. The Eris reasoning rules are then encoded as a small library of proof functions over these resources. To justify the soundness of this encoding, we construct a semantic model that relates it to Eris, just as VerusBelt relates Verus's encoding of permissions to Iris. This analogy between the designs is illustrated in Figure 1. This section describes the encoding. The semantic model is later explained in §6.

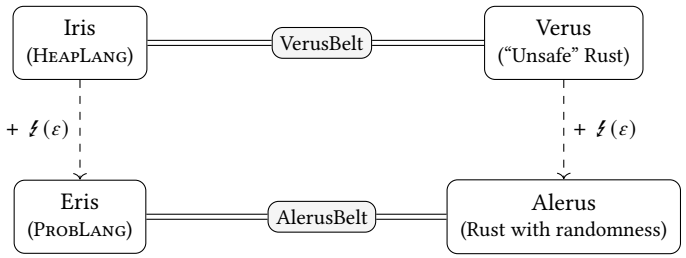


Fig. 1. High-Level Picture of Alerus

3.1 Error Credits in Verus

We obtain error credits by implementing the `ResourceAlgebra` and `PCM` interfaces of §2 for the following carrier type:

$$\mathsf{ErrorCreditCarrier} \ ::= \ \mathsf{Value}(c \in \mathbb{R}) \ \mid \ \mathsf{Empty} \ \mid \ \mathsf{Invalid}$$

A $\mathsf{Value}(c)$ represents an error credit of magnitude c. Note that the Value type ranges over arbitrary real numbers instead of merely non-negative real numbers. While negative error credits should not be representable, we use arbitrary reals so that we can interface with the SMT real theory, which works over arbitrary reals. Meanwhile, Empty is the `PCM` unit, analogous to having zero error credits. Finally, as the name suggests, Invalid is used to represent combinations of error credit resources that are unrepresentable or invalid.

The composition `op`, written as $\cdot$, is defined by

$$\mathsf{Value}(c_1) \cdot \mathsf{Value}(c_2) = \begin{cases} \mathsf{Value}(c_1 + c_2) & \text{if } c_1 \geq 0 \text{ and } c_2 \geq 0, \\ \mathsf{Invalid} & \text{otherwise.} \end{cases}$$

A credit is valid when it is either an empty credit or a value of c for c lying in the range $[0, 1)$:

$$\mathsf{valid}(\mathsf{Value}(c)) \iff 0 \leq c < 1.$$

All error credits live at a single global ghost location. A `tracked ErrorCreditResource` whose view is $\mathsf{Value}(\varepsilon)$ is then analogous to the Eris assertion $↯(\varepsilon)$.

Then, Eris's Err-Split, which allows for credit composition $↯(\varepsilon_1) * ↯(\varepsilon_2) \dashv\vdash ↯(\varepsilon_1 + \varepsilon_2)$, is witnessed by the proof functions `ec_combine` and `ec_split`, which respectively merge two credits and split one credit into two summands. These have the following signatures:

```
pub proof fn ec_combine(
  tracked c1: ErrorCreditResource,
  tracked c2: ErrorCreditResource,
  v1: real, v2: real,
) -> (tracked out: ErrorCreditResource)
  requires
    c1@ =~= Value { car: v1 },
    c2@ =~= Value { car: v2 },
    v1 >= 0real, v2 >= 0real,
  ensures
    out@ =~= Value { car: v1 + v2 },
```

```
pub proof fn ec_split(
  tracked c: ErrorCreditResource,
  v1: real, v2: real,
) -> (tracked (c1, c2):
  (ErrorCreditResource, ErrorCreditResource))
  requires
    c@ =~= Value { car: v1 + v2 },
    v1 >= 0real, v2 >= 0real,
  ensures
    c1@ =~= Value { car: v1 },
    c2@ =~= Value { car: v2 },
```

Both are pure `proof` functions, so they manipulate only ghost state and are erased after verification. The `ec_combine` function consumes two `tracked` resources whose views, represented by Verus's postfix `@` operator, are Value(v_1) and Value(v_2) and returns a single resource whose view is Value($v_1 + v_2$), realizing the right-to-left direction of Err-Split. Meanwhile, `ec_split` runs this in reverse: it consumes one resource of view Value($v_1 + v_2$) and hands back a pair of resources with views Value(v_1) and Value(v_2). These are `Resource`'s `join` and `split` functions for this instance. In both directions the $v_i \geq 0$ preconditions mirror the non-negativity side condition built into op, so that the credits being merged or divided are always valid; the $=\!\sim\!=$ relation is Verus's extensional equality on views.

Next, we get something analogous to Err-1, which says $\lightning(1) \vdash \bot$, through the proof function `ec_contradict`. This derives a contradiction from any credit of value greater than or equal to one, using `Resource`'s `validate` to learn that the held value is `valid`. Here, `ec_contradict` takes only a shared reference to a credit resource, since it does not consume the credit but merely inspects it. Its precondition asserts that the resource's view is some Value(car) with $car \geq 1$, and from this it derives false. The proof of this follows from the fact that Verus already has a rule saying that ownership of an invalid element implies false. Since a credit is valid only when its value lies in $[0, 1)$, ownership of a credit of value ≥ 1 thus implies false.

```
pub proof fn ec_contradict(
  tracked e: &ErrorCreditResource,
)
  requires
    exists |car: real| car >= 1
    real
      && e@ =~= Value { car },
  ensures
    false,
```

3.2 Axiomatized Eris Rules in Verus

The algebraic rules for error credits shown above are proved from the resource definition. The two remaining Eris rules, expectation preservation (Err-Rand-Exp) and thin air (Err-Thin-Air), are the only constructs Alerus adds as trusted axioms. Both are axiomatized as executable functions acting on the credit resource. This is similar to how Verus must axiomatize the connection between points-to resources and the underlying Rust commands that mutate physical state, such as `ptr_mut_ref`.

Expectation Preservation. To encode the expectation-preserving rule, we first need to decide on the primitive that we want to use for drawing random samples. In our examples, we use a single primitive for drawing random numbers, `rand_ubig`, which generates random samples and returns values of type `UBig`, which represents an arbitrary-precision unsigned integer. We also use `rand_ubig` to derive `rand_u64`, which generates uniform random samples over 64-bit unsigned integers.[2] The bignum implementation is from the `dashu` crate [51]. We give the bignum operations trusted specifications, since verifying the bignum library itself is beyond the scope of this paper.

[2]For performance reasons, you might want to have `rand_u64` as a standalone axiomatized primitive, but we do not do so here for simplicity.

```
#[verifier::external_body]
pub fn rand_ubig(
  bound: &UBig,
  Tracked(e1): Tracked<ErrorCreditResource>,
  Ghost(e2): Ghost<spec_fn(nat) -> real>,
) -> ((n, out_credit): (UBig, Tracked<ErrorCreditResource>))
  requires
    ubig_view(bound) > 0,
    forall |i: nat| e2(i) >= 0real,
    exists |eps: real| e1@ =~= Value { car: eps }
      && eps >= average_nat(ubig_view(bound), e2),
  ensures
    ubig_view(&n) < ubig_view(bound),
    out_credit@@ =~= Value { car: e2(ubig_view(&n)) },
```

The function `rand_ubig(bound, e1, e2)` samples a value n uniformly from $\{0, \ldots, N-1\}$, where $N =$ `ubig_view(bound)` is the arbitrary-precision bound. The `tracked` argument `e1` carries the input credit of value ε, and the `Ghost` argument `e2` is the credit-allocation function $\mathcal{E} : \mathbb{N} \to \mathbb{R}$. The precondition requires $\mathtt{e2}(i) \geq 0$ for every i, together with the expectation-preservation side condition that the held credit dominate the uniform average of `e2`, computed by `average_nat`: $\varepsilon \geq \mathtt{average\_nat}(N, \mathtt{e2}) = \frac{1}{N} \sum_{i<N} \mathcal{E}(i)$. Just as in Err-Rand-Exp, the function returns the sample `n` together with the output credit `out_credit`, and the postcondition guarantees that this credit has value $\mathcal{E}(n)$.

Thin Air. The `thin_air()` axiom returns a `tracked` credit of *some* value $\varepsilon > 0$ out of thin air. Note that our encoding of this axiom deviates slightly from the way the rule Err-Thin-Air was phrased: rather than quantifying over all possible $\varepsilon > 0$ and requiring that the Hoare triple be proved for each ε, we instead existentially quantify over some $\varepsilon > 0$ and add that to the context. This formulation is logically equivalent, and is a better fit because Verus doesn't have impredicative Hoare triples, so there's no way for us to write a rule that requires a collection of Hoare triples to be proved.

```
#[verifier::external_body]
pub fn thin_air() -> (ret: Tracked<ErrorCreditResource>)
  ensures
    exists |eps: real| eps > 0
      && ret@@ =~= Value { car: eps },     // owns a credit eps > 0
```

One other difference between the formulation of this rule in Verus and the version in Eris is that this version requires that the program be explicitly annotated with the invocation of this ghost operation. In contrast, in Eris, the proof rule can be invoked at any point, without annotating the program. In our examples, this difference is not a limitation, because there is usually a clear point where one wishes to invoke the rule, making it easy to add this annotation.

3.3 Example

We now put these pieces together in a small self-contained example. Consider a program that flips two fair coins using `rand_2_u64`, and returns whether both came up heads, where heads is encoded as 1. We will prove that, given $1/4$ error credit up front, the function returns `false`. In other words, the outcome where it would return `true` occurs with probability at most $1/4$.

```
pub fn flip_and(Tracked(credit): Tracked<ErrorCreditResource>) -> (ret: bool)
  requires
    credit@ =~= (Value { car: 1real / 4real }),
  ensures
    ret == false,
{
  let (b1, Tracked(c1)) = rand_2_u64(
```

```
    Tracked(credit),
    Ghost(|x: nat| if x == 1 { 1real / 2real } else { 0real })
  );
  let (b2, Tracked(c2)) = rand_2_u64(
    Tracked(c1),
    Ghost(|x: nat| if b1 == 1 && x == 1 { 1real } else { 0real })
  );
  proof { if b1 == 1 && b2 == 1 { ec_contradict(&c2); } }
  (b1 == 1) && (b2 == 1)
}
```

The proof works by picking the error credit allocation ($\mathcal{E}$ in Err-Rand-Exp) for each coin flip so that we get $\lightning(1)$ for the "bad" outcome we need to rule out (both coin flips yielding 1).

For the first flip we hand Err-Rand-Exp the allocation $\mathcal{E}_1$ which gives $\frac{1}{2}$ on outcome 1 and 0 on outcome 0. The average of this allocation is $\frac{1}{4}$, which is exactly the credit we start with. By the postcondition of `rand_2_u64` we then own $\lightning(\mathcal{E}_1(\texttt{b1}))$. For the second flip the allocation $\mathcal{E}_2$ is conditioned on the first coin, paying 1 when `b1` $= 1$ and the new draw is 1, and 0 in all other cases. The average of this allocation is $\frac{1}{2}$ when `b1` $= 1$ and 0 when `b1` $= 0$, which is exactly the credit we own after the first flip. Hence, along the single path `b1` $=$ `b2` $= 1$ we end up owning $\lightning(\mathcal{E}_2(1)) = \lightning(1)$, and on every other path only $\lightning(0)$. The `proof` block invokes `ec_contradict` (Err-1) exactly on the former path, turning the full credit into a proof of `false`, thereby discharging that proof branch. Since the return value $(\texttt{b1} = 1) \wedge (\texttt{b2} = 1)$ is `true` only on that excluded path, every other case returns `false`, discharging the postcondition.

4 Almost Sure Termination with Error Credits

As discussed in §2, Verus enables showing termination via `decreases` clauses. In probabilistic programs, rather than focusing on the usual characterization of termination, it is common to consider *almost sure termination*, *i.e.*, to show that the program terminates with probability 1. To see why this is challenging, consider the following program, which generates samples from the geometric distribution. It repeatedly generates samples from $\{0, 1\}$ uniformly and counts the number of 1s generated before the first 0. The repetition is done by recursively calling `geometric` in the case where a 1 is generated.

This program terminates with probability 1, because the diverging execution that repeatedly samples 1 forever occurs with probability 0. However, we cannot establish termination using a traditional `decreases` clause because there's no obviously decreasing measure to supply. The function takes no arguments, and the recursive call in the `else` branch is made directly, with no quantity that visibly shrinks between calls.

```
pub fn geometric() -> (ret: UBig) {
  let val = rand_2_u64(); // fair coin
  if val == 0 {
    UBig::ZERO      // terminate
  } else {
    geometric() + UBig::ONE // recurse
  }
}
```

To deal with this, previous probabilistic program logics and deductive verifiers based on them [36, 41] encode specialized reasoning rules for almost-sure termination. However, in Alerus, we achieve almost-sure termination reasoning without needing to add any specialized rules. The key is that once we have error credits, we can formulate a decreasing measure that will work with Verus's existing `decreases` clause.

Aguirre et al. [1] call this form of termination reasoning "credit amplification". The idea behind credit amplification is to show that on non-terminating branches, when the program recurses or loops, the amount of credit owned increases and will eventually reach 1. Since $\lightning(1) \vdash \bot$, once the

proof accumulates a credit of value 1, we are done. Under the hood, by accumulating error credits in this fashion, we have effectively shown that the probability of non-termination goes to zero.

The general methodology is to find a way to bound the number of steps or repetitions that it will take to reach $\lightning(1)$, and then use that number of steps as the measure to supply to the `decreases` clause. Because the number of steps is some finite number that will decrease, this works with Verus's existing mechanisms for reasoning about termination through the `decreases` clause.

We showcase this technique first by proving almost sure termination of the geometric sampler. We reformulate the program below in order to thread the error-credit resource through it. The entry point `geometric` owns no error credit yet, so it invokes `thin_air ()` to materialize a credit of arbitrary value $\varepsilon > 0$ as the ghost resource `e`. On each iteration, we will redistribute the credits so that the terminating branch receives 0 credits, and the non-terminating branch obtains twice the amount of credit. Thus, if we start with ε error credits, then after n iterations, we will have $2^n \cdot \varepsilon$ error credits. The proof uses the Archimedean property of the reals to pick an n such that $\varepsilon \cdot 2^n \geq 1$, and hands both ε and n to a recursive helper function `bounded_geometric`.

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

The helper `bounded_geometric` does the actual sampling. The invariant $\varepsilon \cdot 2^{\mathsf{depth}} \geq 1$ is carried in the precondition, and termination is justified by **decreases** `depth`; when `depth` $= 0$ the invariant forces $\varepsilon \geq 1$. Each iteration calls `rand_2_u64` with a credit allocation that assigns 0 on the terminating branch and 2ε on the recursing branch, which satisfies Err-Rand-Exp as $(0 + 2\varepsilon)/2 = \varepsilon$. So ε is amplified to 2ε and forwarded to the recursive branch. There, $2\varepsilon \cdot 2^{\mathsf{depth}-1} = \varepsilon \cdot 2^{\mathsf{depth}} \geq 1$, restoring the invariant at the smaller depth and closing the recursion.

```
pub fn bounded_geometric(Tracked(in_credit): Tracked<ErrorCreditResource>, Ghost(depth): Ghost<nat>) -> UBig
  requires exists |eps: real| {
    &&& eps > 0real
    &&& in_credit@ =~= (Value { car: eps })
    &&& eps * pow(2real, depth) >= 1real
  },
  decreases depth,
{
  /* ... get epsilon and show depth is non-zero ... */
  let (val, Tracked(out_credit)) = rand_2_u64(Tracked(in_credit),
    Ghost(|x: nat| if x == 0 { 0real } else { 2real * eps }), // 0 |-> 0, 1 |-> 2*eps
  );
  if val == 0 {
    UBig::ZERO // terminating branch
  } else {
    // 2*eps * 2^(depth-1) = eps * 2^depth >= 1, invariant restored
    bounded_geometric(Tracked(out_credit), Ghost((depth - 1) as nat)) + UBig::ONE
  }
}
```

Beyond geometric loops. The almost-sure termination reasoning for the geometric sampler is quite simple: the credit consistently doubles in the non-terminating branch. To show we can also handle nontrivial almost-sure termination properties, we next prove the almost-sure termination of a one-dimensional random walk. This example will illustrate a general methodology for using the credit allocation to make the termination proof go through.

The one-dimensional random walk is a probabilistic program that starts at any position $n \in \mathbb{N}$; at each step, it increases the position by 1 or decreases it by 1 with equal probability. The program terminates when it first hits 0. Unlike in the case of the geometric sampler, here we cannot pick a constant amplification factor for the error credits, since the walker can drift arbitrarily far from the origin before returning.

The key is that the credit allocation function we hand to `rand_2_u64`, rather than amplifying the credit by a fixed amount, now depends on the current position. How do we define what this function should be? Recall that we ultimately want to have some "fuel" parameter s that decreases on each recursive call or loop iteration such that when $s = 0$, the program terminates. For a given value of s, we can think of the credit allocation function as needing to distribute credits so that when the walker ends up in state p, we receive credits equal to the probability that the program *does not* terminate after s rounds starting from p.

```
1  pub fn random_walk(pos: UBig) -> (ret: UBig) {
2    if pos == UBig::ZERO {  // at the origin
3      UBig::ZERO
4    } else {
5      let val = rand_2_u64(); // fair coin
6      if val == 0 {
7        random_walk(pos - UBig::ONE)
8      } else {
9        random_walk(pos + UBig::ONE)
10     }
11   }
12 }
```

We can define this function $\texttt{fail\_prob}(s, p)$ in terms of a recurrence relation:

$$\begin{aligned} \texttt{fail\_prob}(s, 0) &= 0, \\ \texttt{fail\_prob}(0, p) &= 1 \quad \text{for } p > 0, \\ \texttt{fail\_prob}(s, p) &= \tfrac{1}{2}\big(\texttt{fail\_prob}(s{-}1,\, p{-}1) + \texttt{fail\_prob}(s{-}1,\, p{+}1)\big). \end{aligned} \tag{1}$$

This definition is well-founded because the parameter s decreases on each recursive call, so we can define it as a valid `spec` function in Verus. The first case is when we successfully reach 0 and terminate. The second case is when we run out of fuel and have not yet reached 0. In that case, the probability that we will not terminate after 0 more rounds is 1. The third case captures how the random walk updates the position.

Since we have defined `fail_prob` to be the amount of credit we need to ensure termination, if we prove $\forall \delta > 0.\ \exists s.\ \texttt{fail\_prob}(s, \texttt{pos}) < \delta$, then this means that for any positive credit budget δ, there is some s such that we can use it to pay for the program's termination within s steps. Proving this is a pure mathematical fact about the recurrence given by (1). An explanation of this proof can be found in Appendix A. In particular, in Verus, this is a straightforward fact we prove about the `spec` function `fail_prob`.

Once we have this fact, we turn to actually reasoning about the code implementing `random_walk`. We first use the Err-Thin-Air rule to get some arbitrary $\varepsilon > 0$. Using the property just proved for `fail_prob`, we get some fuel s large enough that $\texttt{fail\_prob}(\texttt{s}, \texttt{pos}) < \varepsilon$. The remaining work is to show that, when we update our error credits using Err-Rand-Exp, the resulting credits we use match the cases of the recurrence relation defining `fail_prob`.

The steps of the general pattern are to: (1) define a pure mathematical recurrence relation for the probability of non-termination after s steps, (2) prove that for any $\varepsilon > 0$ there exists s large enough that ε is greater than this probability of non-termination, and (3) generate a thin-air credit and prove a correspondence between the recurrence relation and the code. Isolating the pure mathematical reasoning in steps 1 and 2 from the code reasoning in step 3 helps to structure the proofs. In §5, we will see how to apply this pattern to several challenging case studies.

We have verified several other examples from Aguirre et al. [1], including almost-sure termination of the escaping spline [36] and a higher-order rejection sampler. In fact, it is no accident that this

approach to reasoning about almost-sure termination works so well. Recent work [21] establishes that the Eris program logic is *complete* for almost-sure termination on a higher-order probabilistic language: any program that terminates with probability 1 can, in principle, be proven to do so using error credits by picking the right credit distribution.[3] From a practical standpoint, this gives us confidence that the almost-sure termination support we expose in Verus is not artificially limited to a narrow class of programs. Although the hard part of solving the recurrence relation remains, the rules are expressive enough to cover a wide class of programs.

5 Verifying Sampler Correctness

Now that we have seen how Alerus's encoding of error credits works for verifying some simple bounds and proving almost-sure termination, we turn to proving the correctness of samplers written in Rust. As explained in §2.2, Marionneau et al. [33] proved a soundness theorem for Eris, which showed that, by proving a specification about a program e that allows for error credits to be distributed in a way that preserves expected values under a distribution μ, we can establish that e in fact returns samples according to μ. We call specifications of this form EPT in the rest of this section. Using such specifications, we prove the correctness of a number of samplers, including the discrete Laplace and discrete Gaussian samplers (§5.1), the alias method (§5.2), and the fast loaded dice roller (§5.3).

To illustrate the pattern, we warm up with a simple example: a sampler for the Bernoulli distribution $\mathrm{Bern}(p)$, where p is a rational number of the form a/b. The sampler draws a single uniform value u over $\{0, \ldots, b-1\}$ and returns whether u falls below the numerator a, which happens with probability exactly $a/b = p$.

```
fn sample_bernoulli_rational(a: &UBig, b: &UBig) -> bool {
    let u = rand_ubig(b);   // u ~ Uniform([0, b))
    u < a                   // true with probability a/b
}
```

Proving this sampler correct amounts to proving the following expectation-preserving specification, in the form of EPT, for every caller-supplied credit allocation $\mathcal{E}$ over the two outcomes:

$$\frac{\varepsilon \;\geq\; p\,\mathcal{E}(\texttt{true}) + (1-p)\,\mathcal{E}(\texttt{false})}{\vdash\; [\lightning(\varepsilon)]\ \texttt{sample\_bernoulli\_rational}(a, b)\ [v.\ \lightning(\mathcal{E}(v))]}$$

Concretely, to prove this specification, we are given an $\mathcal{E}$ satisfying the inequality in the premise, and a precondition of $\lightning(\varepsilon)$ error credits. We must figure out, based on $\mathcal{E}$, what credit allocation function f to supply to the call of rand_ubig. To do so, we essentially perform backwards reasoning, determining what values of u cause sample_bernoulli_rational to return `true` and `false`. From this, we get the requirement that $f(\mathsf{u}) = \mathcal{E}(\texttt{true})$ for $\mathsf{u} < a$ and $f(\mathsf{u}) = \mathcal{E}(\texttt{false})$ for $\mathsf{u} \geq a$. We then check that this choice of f satisfies the precondition for rand_ubig, which follows since

$$\tfrac{1}{b}\textstyle\sum_{u=0}^{b-1} f(u) \;=\; \tfrac{1}{b}\big(a\,\mathcal{E}(\texttt{true}) + (b-a)\,\mathcal{E}(\texttt{false})\big) \;=\; p\,\mathcal{E}(\texttt{true}) + (1-p)\,\mathcal{E}(\texttt{false}) \;\leq\; \varepsilon. \quad (2)$$

The postcondition of rand_ubig then gives us the appropriate number of credits to prove the postcondition.

General Proof Strategy. The general recipe for proving sampler correctness is to do a form of backwards reasoning. Letting *ret* be the final return value of the sampler function, we trace backwards to find a symbolic expression for $\mathcal{E}(ret)$ in terms of the intermediate random samples the code generates to compute *ret*. This formula determines the shape of the credit allocation

[3] Hostert et al. [21] restrict to programs without dynamic allocation because Eris's rules are incomplete for reasoning about the memory addresses an allocator will assign, but this is orthogonal to reasoning about termination probabilities.

for each intermediate sampler call as we work backwards. Tracing back to the beginning of the sampler routine, we prove a mathematical fact that the overall symbolic formula is bounded by the expectation of $\mathcal{E}$ under the distribution that the sampler is intended to generate.

If the sampler involves loops or recursion, the symbolic formula for the credit allocation will naturally be structured as a recurrence relation, which we solve or bound by induction. For these samplers we also have to prove termination using a `decreases` clause. To handle this, we use a *separate* supply of error credits, generated from an initial thin-air credit and amplified using the approach described in §4. This separates out the termination reasoning from reasoning about whether each sample has the correct weight.

In addition, we structure our Verus proofs so that all of the pure mathematical reasoning, such as bounding a recurrence relation or proving that the symbolic formula is algebraically equivalent to the appropriate expected value, is isolated to separate lemmas. The core reasoning about the executable code then just reduces to proving that the symbolic formula accurately reflects the credit transformations needed at intermediate sampler calls. A benefit of this decomposition is that the pure mathematical facts are easily discharged by a combination of Verus's SMT automation and LLM agents. In the following examples, all of the Verus proofs for intermediate mathematical lemmas were done by Claude Opus 4.7 and 4.8, without human intervention.

5.1 Discrete Gaussian

We prove the correctness of the discrete Laplace and discrete Gaussian samplers implemented in OpenDP [45], which use the CKS algorithm [9].

The samplers are verified in three phases: first, we verify a sampler for $\mathrm{Bern}(e^{-x})$ for $x \geq 0$; then we use it to build two different samplers for $\mathrm{Geom}(1-e^{-x})$; finally, we use the geometric sampler to build the discrete Laplace and discrete Gaussian samplers. All of these algorithms rely only on the single primitive sampler `rand_ubig` we axiomatized in §3.2. Each sampler is verified against an EPT specification and reused as a sub-sampler by the next, forming the call graph in Figure 2.

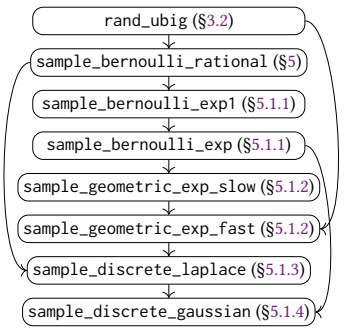


Fig. 2. Call graph of the CKS algorithm for the discrete Gaussian: an arrow $A \to B$ means A is used as a sub-sampler by B.

5.1.1 Negative Exponential Bernoulli. We start with a sampler for the negative exponential Bernoulli distribution $\mathrm{Bern}(e^{-x})$. We represent x as a rational number with `RBig` in Rust, which is an arbitrary rational number type, which can be broken down into the numerator and denominator as `IBig`/`UBig`. First, we implement `sample_bernoulli_exp1`, which imposes the restriction that $x \in [0, 1]$. This is then called by `sample_bernoulli_exp`, which drops the upper bound restriction.

The sampler flips $\mathrm{Bern}(x/k)$ for increasing $k = 1, 2, \ldots$, and as soon as a flip returns false, it returns whether the current k is odd. In order to show that this code samples from $\mathrm{Bern}(e^{-x})$, it suffices to prove an EPT specification of the following form, where $\mathcal{E}$ is the caller-supplied credit allocation.

$$\frac{\varepsilon \;\geq\; e^{-x}\,\mathcal{E}(\mathtt{true}) + (1-e^{-x})\,\mathcal{E}(\mathtt{false})}{\vdash\ [\lightning(\varepsilon)]\ \mathtt{sample\_bernoulli\_exp1}(x)\ [b.\ \lightning(\mathcal{E}(b))]}$$

```
fn sample_bernoulli_exp1(x: RBig) -> bool {
  let mut k = UBig::ONE;
  loop {
    if sample_bernoulli_rational(&x, &k) {
      k += UBig::ONE;
    } else { return k.is_odd(); }
  }
}
```

We prove this specification by reasoning backwards from the credit allocation $\mathcal{E}$ to construct the credit allocation we hand to each sub-sampler, and then discharge that sub-sampler's EPT rule. The one new challenge is that `sample_bernoulli_exp1` loops rather than making a single draw, so we have to maintain a loop invariant to save enough credits for each draw. This invariant keeps track of the credit ε_k on entry to the kth iteration, ensuring it suffices to cover the conditional expectation of $\mathcal{E}$ over the eventual result,

$$\varepsilon_k \;\geq\; p_k\,\mathcal{E}(\texttt{true}) + (1-p_k)\,\mathcal{E}(\texttt{false}),$$

where p_k is the probability that the sampler eventually returns `true` when the loop has reached iteration k. We get the following recurrence by conditioning on the outcome of the step-k flip $\mathrm{Bern}(x/k)$, which continues to step $k+1$ with probability x/k and otherwise stops and returns whether k is odd.

$$p_k \;=\; \tfrac{x}{k}\,p_{k+1} + \left(1-\tfrac{x}{k}\right)[k \text{ odd}], \qquad p_1 = e^{-x}.$$

Here, $[k \text{ odd}]$ denotes the Iverson bracket, which is 1 when k is odd and 0 otherwise. At iteration 1 the invariant is exactly the caller's precondition, since $p_1 = e^{-x}$. The only analytic fact the argument needs about the p_k is that each is a genuine probability in $[0,1]$, which follows from an alternating Taylor-series bound on e^{-x}.

Reasoning backwards through the step-k call to `sample_bernoulli_rational` tells us the credit allocation function must be

$$b \;\mapsto\; \begin{cases} \varepsilon_{k+1} & b = \texttt{true} \;\; \text{(continue)},\\ \mathcal{E}([k \text{ odd}]) & b = \texttt{false} \;\; \text{(return k.is\_odd())}, \end{cases} \qquad \varepsilon_{k+1} \;=\; \tfrac{k}{x}\,\varepsilon_k - \left(\tfrac{k}{x}-1\right)\mathcal{E}([k \text{ odd}]),$$

where ε_{k+1} is solved for precisely so that `sample_bernoulli_rational`'s EPT holds. The rest of the proof checks that the credit allocation preserves the loop invariant at $k+1$, which is a rearrangement of terms. We omit this here; further details can be found in Appendix B.

Modeling e^{-x}. Z3 does not natively support transcendental functions like the natural exponential. We therefore expose $e^{(-)}$ as an uninterpreted function in Verus and axiomatize the properties we need. The key Taylor-tail bound used above is the only nontrivial axiom, and is described in Appendix C; we verify it separately in Lean.

Generalizing to $\mathrm{Bern}(e^{-x})$ for $x \geq 0$. We next extend the sampler to arbitrary nonnegative x using `sample_bernoulli_exp`, which calls the previous `sample_bernoulli_exp1`. This sampler performs $\lfloor x \rfloor$ independent $\mathrm{Bern}(e^{-1})$ flips, returning false as soon as any of them fails, and if all succeed, it returns a final $\mathrm{Bern}(e^{-\mathrm{frac}(x)})$ flip. All of these flips are independent, so the probability they all return true is given by the product of the probabilities that each individually returns true. This is e^{-x}, since $e^{-x} = e^{-\lfloor x \rfloor} \cdot e^{-(x-\lfloor x \rfloor)}$.

```
fn sample_bernoulli_exp(x: &RBig) -> bool {
  let mut whole = x.floor();
  while whole > IBig::ZERO {
    if !sample_bernoulli_exp1(RBig::ONE) {
      return false;
    }
    whole -= IBig::ONE;
  }
  sample_bernoulli_exp1(x.frac())
}
```

Working backwards, we find that the credit allocation function supplied to the final call to `sample_bernoulli_exp1(x.frac())` is exactly $\mathcal{E}$. For the loop, we establish a similar loop invariant that the held credit dominates the conditional expectation of $\mathcal{E}$. Writing r for the value of `x` still remaining, this invariant says that we have ε error credits with

$$\varepsilon \;\geq\; e^{-r}\,\mathcal{E}(\texttt{true}) + (1-e^{-r})\,\mathcal{E}(\texttt{false}).$$

Concretely, to the iteration's `sample_bernoulli_exp1(1)` call, we give the credit allocation F:

$$F(b) \;=\; \begin{cases} e^{-(r-1)}\,\mathcal{E}(\texttt{true}) + (1-e^{-(r-1)})\,\mathcal{E}(\texttt{false}) & b = \texttt{true} \;\; (\text{continue},\; r \mapsto r-1), \\ \mathcal{E}(\texttt{false}) & b = \texttt{false} \;\; (\text{return false}). \end{cases}$$

On `true` it forwards the invariant's credit for the smaller problem $e^{-(r-1)}$, and on `false` it pays $\mathcal{E}(\texttt{false})$. The rest is to check that the EPT for `sample_bernoulli_exp1(1)` preserves the loop invariant:

$$e^{-1}\,F(\texttt{true}) + (1-e^{-1})\,F(\texttt{false}) \;=\; e^{-r}\,\mathcal{E}(\texttt{true}) + (1-e^{-r})\,\mathcal{E}(\texttt{false}).$$

5.1.2 Geometric Bernoulli. We build a $\mathrm{Geom}(1-e^{-x})$ sampler from the negative exponential Bernoulli sampler, and it has a fast and a slow version. The slow version is a standard geometric sampler, similar to the one in §4. It repeatedly draws $\mathrm{Bern}(e^{-x})$ until it returns false, and returns the number of draws that returned true. The fast version calls the slow version as a sub-sampler. We omit the verification of the slow version and focus on the fast one.

Write $x = n/d$ in lowest terms, with n and d positive integers. The sampler draws two independent samples and combines them by integer division. The exponential rejection sampler (`sample_exp_rejection`) repeatedly draws u uniformly from $\{0,\dots,d-1\}$ and keeps it with probability $e^{-u/d}$, so it returns u with probability $e^{-u/d}/N$, where $N = \sum_{u=0}^{d-1} e^{-u/d}$. Then it calls the "slow" geometric sampler $v \sim \mathrm{Geom}(1-e^{-1})$, and returns $\lfloor (u + d\,v)/n \rfloor$.

```rust
fn sample_exp_rejection(d: &UBig) -> UBig {
  loop {
    let u = sample_uniform_ubig_below(d);
    if sample_bernoulli_exp_ubig(&u, d) {
      return u;
    }
  }
}
```

```rust
pub fn sample_geometric_exp_fast(x: RBig) -> UBig {
  let ((_, n), d) = x.into_parts(); // x = n/d
  let u = sample_exp_rejection(&d);
  let v = sample_geometric_exp_slow(&RBig::ONE);
  (v * d + u) / n
}
```

Aside from the integer floor divisions, the structure here is straightforward, and the credit reasoning is mechanical.

We first prove the exponential-rejection sampler's EPT specification:

$$\frac{N = \sum_{u=0}^{d-1} e^{-u/d} \qquad \varepsilon \;\geq\; \mathbb{E}_{u\sim\text{rejection}}[\,\mathcal{E}(u)\,] \;=\; \frac{1}{N}\sum_{u=0}^{d-1} e^{-u/d}\,\mathcal{E}(u)}{\vdash\; [\lightning(\varepsilon)]\;\texttt{sample\_exp\_rejection}(d)\;[u.\;\lightning(\mathcal{E}(u))]}$$

Since it's a stateless rejection sampler, the loop invariant preserves the ε credits we started with, as each iteration is independent. This forces the allocation function for $\mathrm{Bern}(e^{-u/d})$ to pay out $\mathcal{E}(u)$ on `true` and carry ε back to the next iteration on `false`. And the credit allocation given to the uniform draw of u is the $\mathrm{Bern}(e^{-u/d})$-average,

$$h(u) \;=\; e^{-u/d}\,\mathcal{E}(u) + (1-e^{-u/d})\,\varepsilon.$$

Averaging over the uniform draw and using the precondition $\varepsilon \geq \frac{1}{N}\sum_{u=0}^{d-1} e^{-u/d}\,\mathcal{E}(u)$,

$$\tfrac{1}{d}\sum_{u=0}^{d-1} h(u) \;=\; \tfrac{N}{d}\Big(\tfrac{1}{N}\sum_{u=0}^{d-1} e^{-u/d}\,\mathcal{E}(u)\Big) + \big(1-\tfrac{N}{d}\big)\varepsilon \;\leq\; \varepsilon,$$

so the held credit ε is preserved across iterations. The almost sure termination is given by the thin_air credit, as each iteration amplifies it by a fixed factor $\frac{1}{1-N/d} = \frac{d}{d-N} > 1$.

The right-hand program is then straight-line, and the EPT credit allocations compose: working backwards, the slow geometric sampler is handed $g(u,v) = \mathcal{E}\big(\lfloor (u+d\,v)/n \rfloor\big)$, which forces the

rejection sampler's allocation $f(u) = \mathbb{E}_{v\sim\mathrm{Geom}(1-e^{-1})}[\,g(u,v)\,]$. The remaining proof obligation is then to check that the precondition has enough credits to cover the credit allocation for the exponential rejection sampler, namely $\sum_{r=0}^{\infty}(e^{-n/d})^r(1-e^{-n/d})\,\mathcal{E}(r) \geq \mathbb{E}_{u\sim\mathrm{rejection}}[\,f(u)\,]$. The proof is a rearrangement of two series, using a bijection between the terms, which we defer to Appendix D.

5.1.3 Discrete Laplace. The discrete Laplace $\mathrm{Lap}(0, \mathit{scale})$ is symmetric around 0: writing $p = e^{-1/\mathit{scale}}$, it has probability mass function $\mu_L(x) = \frac{1-p}{1+p}\,p^{|x|}$ for $x \in \mathbb{Z}$. [4]

The sampler flips a fair coin for the sign and draws the magnitude from $\mathrm{Geom}(1-p)$. It rejects when the magnitude is 0 and the sign is negative so that we do not double-count 0.

```
pub fn sample_discrete_laplace(scale: &RBig) -> IBig {
  loop {
    let positive = sample_bernoulli_rational(1, 2);         // fair sign
    let k: IBig = sample_geometric_exp_fast(scale.recip()).into(); // |y| ~ Geom(1-p)
    if positive || !k.is_zero() {    // reject only the (-, 0) duplicate, then retry
      return if positive { k } else { -k };
    }
  }
}
```

This is essentially a rejection sampler wrapping the previous geometric sampler. Thus, we can verify it similarly to the exponential-rejection sampler (§5.1.2). Reading the allocation off the program backwards, the loop maintains $\varepsilon = \sum_{x\in\mathbb{Z}} \mu_L(x)\,\mathcal{E}(x)$, and backward execution through the fair sign flip $\mathrm{Bern}(\frac{1}{2})$ splits the held credit into a positive and a negative branch budget $\varepsilon_+, \varepsilon_-$ (the negative branch carrying ε back on the rejected $(-,0)$ outcome). Discharging the sign flip's EPT precondition then reduces to $\varepsilon_+ + \varepsilon_- \leq 2\,\varepsilon$, which we verify in Appendix E.

5.1.4 Discrete Gaussian. The discrete Gaussian $\mathcal{N}_{\mathbb{Z}}(0,\sigma^2)$ is sampled by rejection against a discrete-Laplace proposal: with $t = \lfloor\sigma\rfloor + 1$, we draw $y \sim \mathrm{Lap}(0,t)$ and accept it with probability $e^{-\mathrm{bias}(y)}$, where $\mathrm{bias}(y) = (|y| - \sigma^2/t)^2/(2\sigma^2)$. Its verification uses the same rejection-sampler technique as the discrete Laplace. We defer the sampler and its correctness proof to Appendix F.

5.2 Alias Method

From the previous subsection, we see that Alerus can verify samplers with intricate probabilistic behavior. In this subsection and the next, we turn to examples that combine probabilistic reasoning with data structures built from mutable state. These demonstrate the benefit of Alerus's ability to build on Verus's existing support for reasoning about stateful data structures.

We start with the alias method for sampling from finite discrete distributions [42, 46, 47]. Given a finite set of n labels $\{0,\ldots,n-1\}$ and a vector of nonnegative integer weights $(a_0, a_1, \ldots, a_{n-1})$, the alias method samples from the distribution that returns label i with probability $a_i/\sum_{j=0}^{n-1} a_j$.

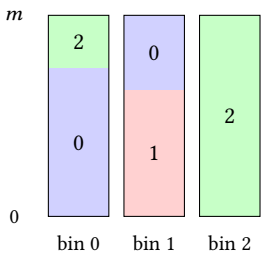


Fig. 3. The alias table for weights $(a_0, a_1, a_2) = (7, 4, 8)$.

The algorithm is split into two functions: a preprocessing function and a sampling function. The preprocessing function returns an *alias table* that is used for subsequent sampling. The table consists of two length-n arrays, *prob* and *alias*. Picture n equal-probability *bins*, one per index, each filled with a total of m units of mass. Bin i is split between at most two labels: it holds $\mathit{prob}[i]$ units of its

[4] This looks different from the standard Laplace density: $\frac{1}{2\sigma}\,e^{-|x|/\sigma}$ for $\sigma = \mathit{scale}$. Since $p = e^{-1/\sigma}$, $\mu_L(x) = \frac{e^{1/\sigma}-1}{e^{1/\sigma}+1}\,e^{-|x|/\sigma}$, with the same $e^{-|x|/\sigma}$, and the normalizer sums to $\sum_{x\in\mathbb{Z}} e^{-|x|/\sigma} = \frac{1+p}{1-p}$.

own label i and the remaining $m - prob[i]$ units of an *alias* label $alias[i]$. Summing over all bins shows that each label k owns exactly $n\,a_k$ units. Figure 3 shows an alias table for the weights $(a_0, a_1, a_2) = (7, 4, 8)$. Each bin has $m = 19$ units, with $prob = [14, 12, 19]$, $alias = [2, 0, 2]$. For example, bin 0 holds $prob[0] = 14$ units of label 0 and the remaining 5 units of label 2.

Once the table has been constructed, the sampling routine starts by picking a bin i uniformly. Then, it samples a second value from $\{0, \ldots, m-1\}$, corresponding to a unit in the bin, and returns the label for that unit.

Because the distribution of the samples returned by the sampling function depends on the preprocessed table it is given as input, the specification of the sampler is *parametric* over the table's weights. The table is represented by the executable `AliasTable`, and the sampler's only assumption about it is a well-formedness predicate wf.

```
pub struct AliasTable {
  pub n: u64,   // # of labels
  pub m: u64,   // total weight
  pub weights: Vec<u64>, // a_0..a_{n-1}
  pub prob: Vec<u64>,  // own units per bin
  pub alias: Vec<u64>,  // alias label per bin
}
```

```
pub struct Alias {  // ghost view of AliasTable
  pub n: nat,
  pub m: nat,
  pub weights: spec_fn(nat) -> nat,
  pub prob: spec_fn(nat) -> nat,
  pub alias: spec_fn(nat) -> nat,
}
```

Each executable `AliasTable` has a ghost view `self@ : Alias`, in which the concrete **u64** fields become **nat** and the **Vec<u64>** arrays become mathematical functions `spec_fn(nat) -> nat`. The well-formedness predicate wf requires that the vectors have length `n` and that the ghost view satisfies the key condition `valid_alias(self@)`, a pure predicate on the `Alias` view:

```
  pub open spec fn valid_alias(t: Alias) -> bool {
    &&& t.n >= 1
    &&& t.m >= 1
    &&& t.m == sum_of_weights(t, t.n)
    &&& (forall |i: nat| i < t.n ==> (t.prob)(i) <= t.m)
    &&& (forall |i: nat| i < t.n ==> (t.alias)(i) < t.n)
    // every label k's total units equal n*a_k  (Vose's redistribution invariant).
    &&& (forall |k: nat| k < t.n ==> label_units(t, t.n, k) == t.n * (t.weights)(k))
}
```

The first 5 conjuncts are nondegeneracy conditions, ensuring `AliasTable` has the right shape and that the *prob* and *alias* arrays are well-formed. The last conjunct ensures that the relative weights are correct: for every label k, $label\_units(t, n, k) = n\,a_k$, where *label_units* sums the units of label k across all n bins. This encodes that we have faithfully redistributed the weights into the bins, and is the key invariant that ensures the sampler returns label k with probability a_k/m.

```
pub fn alias_preprocess(weights: Vec<u64>, m: u64)
 -> (ret: AliasTable)
requires
  weights@.len() >= 1,
  m >= 1,
  (weights@.len() as nat) * (m as nat) <= u64::MAX as nat,
  seq_sum(weights@, weights@.len() as nat) == m as nat,
ensures
  wf(ret),
  ret.n as nat == weights@.len(),
  ret.m == m,
  ret.weights@ == weights@,
{
  /* body and proof omitted */
}
```

```
pub fn sample_alias(
  tab: &AliasTable,
  Ghost(e): Ghost<spec_fn(nat) -> real>,
  Tracked(input_credit): Tracked<ErrorCreditResource>,
  Ghost(eps): Ghost<real>,
) -> ((value, out_credit): (u64, Tracked<ErrorCreditResource>))
  requires
    wf(tab),
    forall |x: nat| e(x) >= 0real,
    eps >= alias_exp(tab@, e),
    input_credit@ =~= Value { car: eps },
  ensures
    out_credit@@ =~= Value { car: e(value as nat) },
{
  let i = rand_u64(tab.n);  // bin i ~ U{0..n}
  let r = rand_u64(tab.m);  // slot r ~ U{0..m}
  if r < tab.prob[i] { i }  // own-label part
  else { tab.alias[i] }   // alias part
}
```

The preprocessing function `alias_preprocess` constructs the table and ensures it is well-formed in the postcondition. Meanwhile, `sample_alias` assumes well-formedness as a precondition. In `sample_alias`, the expected value of the user's credit allocation function `e` is computed with `alias_exp`, which uses the distribution represented by the table.

The credit allocation of the sampler itself is straightforward. Conditioned on the drawn bin, the inner threshold draw is expectation-preserving and returns $\lightning(\mathcal{E}(result))$, so we fund the outer bin draw with its per-bin average; regrouping those credits by label and applying the table's validity ($label\_units(t, n, k) = n \cdot weights(k)$) shows the sampler meets the precondition $\varepsilon \geq \sum_{k<n} \frac{a_k}{m} \mathcal{E}(k)$.

Indeed, much of the challenge lies in showing that the preprocessing step constructs the table correctly. We verify an $O(n)$ version [46] that manages two worklists to find labels that still have unaccounted-for units during construction. To the best of our knowledge, there is no prior work on verifying the alias method, as it involves intricate reasoning between preprocessing and sampling. This is precisely the modularity that error credits buy us. The deterministic preprocessing step that builds a well-formed table is verified with ordinary functional-correctness specifications, while the probabilistic sampling step just updates credits, relying on the table being correctly structured through the well-formedness assumption.

5.3 Fast Loaded Dice Roller

Like the alias method, the fast loaded dice roller samples from finite discrete distributions through a preprocessing stage that builds a table and a sampling stage that uses it. However, unlike the alias method, it uses only a fair coin [39]. The algorithm has two stages: a preprocessing stage that constructs a table representing a discrete distribution generating (DDG) tree, and a sampling stage that performs a random walk through this tree. We first focus on a simpler version that only samples from a uniform distribution, called the fast dice roller algorithm.

5.3.1 Fast Dice Roller. The fast dice roller [32] samples uniformly from $\{0, 1, \ldots, n-1\}$ using only fair coin flips. It maintains a state (v, c), where v is the size of what is called the current *window* and c is uniform over $\{0, 1, \ldots, v-1\}$; it starts from the trivial window $v = 1, c = 0$. Each iteration doubles the window and refines c with one fresh coin bit (lines 5-6), which keeps c uniform on $\{0, \ldots, v-1\}$. The window grows until $v \geq n$ (line 7), at which point c is uniform on a range of size $v \geq n$. If $c < n$, the value already lies in the target range and is returned (line 8), uniform on $\{0, \ldots, n-1\}$ as required. Otherwise the algorithm repeats the loop, shifting both down by n (line 9).

```
1  pub fn sample_fdr(n: u64) -> u64 {
2    let mut v: u64 = 1;
3    let mut c: u64 = 0;
4    loop {
5      v = 2 * v;
6      c = 2 * c + rand_2_u64();
7      if v >= n {
8        if c < n { return c }
9        else { v = v - n; c = c - n }
10     }
11   }
12 }
```

The correct credit allocation function to pass to the call to `rand_2_u64` depends on the current state of (v, c). Letting $\mathcal{E}$ be the user-supplied credit allocation function, we define two mutually recursive functions, which are additionally indexed by a fuel parameter k.

$$
\begin{aligned}
&\mathrm{fdr}_f(v, c, 0) = 0 && && \text{(ran out of fuel } k\text{)}\\
&\mathrm{fdr}_f(v, c, k) = \tfrac{1}{2}\big(\mathrm{fdr}_h(2v,\ 2c,\ k{-}1) + \mathrm{fdr}_h(2v,\ 2c{+}1,\ k{-}1)\big) && &&\\
&\mathrm{fdr}_h(v, c, k) = \mathcal{E}(c) && \text{if } v \geq n,\ c < n && \text{(accept)}\\
&\qquad\qquad \mathrm{fdr}_f(v{-}n,\ c{-}n,\ k) && \text{if } v \geq n,\ c \geq n && \text{(reject, restart)}\\
&\qquad\qquad \mathrm{fdr}_f(v,\ c,\ k) && \text{if } v < n && \text{(continue doubling)}
\end{aligned}
$$

These definitions are well-founded since the k parameter decreases by 1 when going from fdr_f to fdr_h. Here $\mathrm{fdr}_f(v, c, k)$ is the conditional expectation $\mathbb{E}[\,\mathcal{E}(out) \mid (v, c)\,]$ of the caller-supplied allocation over the returned value, taken over the next k coin flips from the state (v, c): it models the first `flip` step in the loop. fdr_h models the post-doubling test: mimicking the program structure, it pays out $\mathcal{E}(c)$ on the accepting branch ($v \geq n$, $c < n$), restarts the doubling on the rejecting branch with the shifted window ($v \geq n$, $c \geq n$), and otherwise keeps doubling ($v < n$).

The heavy lifting lies in showing the following pure mathematical fact on fdr_f at the initial state $(1, 0)$:

$$\mathrm{fdr}_f(1, 0, k) \;\leq\; \tfrac{1}{n} \textstyle\sum_{i<n} \mathcal{E}(i),$$

where the sum on the right-hand side is the expected value of $\mathcal{E}$ under the uniform distribution, as required by the EPT-style specification. We defer this proof to Appendix G.

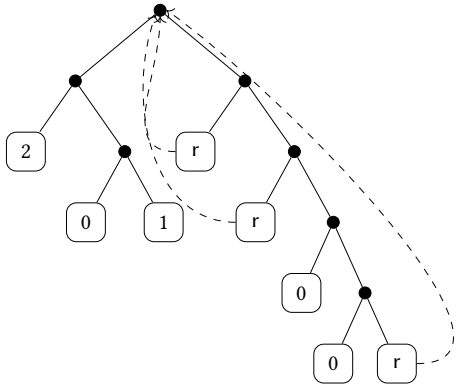
5.3.2 Fast Loaded Dice Roller. The fast loaded dice roller generalizes and builds upon the previous algorithm by sampling from finite discrete distributions where the weights of the choices need not be uniform. In particular, the algorithm is parameterized by a choice of integer weights $a_0, a_1, \ldots, a_{n-1}$ with total $m = \sum_{i=0}^{n-1} a_i$, and the algorithm samples outcome i with probability a_i/m, using only fair coin flips.

The preprocessing step compiles these weights into a discrete distribution generating (DDG) tree: a binary tree whose leaves are labelled by outcomes in $\{0, \ldots, n\}$, where n is an extra label. The sampler walks down the tree one fair coin flip at a time, descending left on 0 and right on 1, tracking the current depth c and node position d, and stopping once it hits a leaf. Starting from the root, the algorithm reaches a leaf at depth c with probability 2^{-c}. If it reaches a leaf labelled with a value in $\{0, \ldots, n-1\}$, it returns that value. Otherwise, on reaching a leaf with label n, it returns to the root of the tree and repeats.

To construct the tree, the first step is to assign a weight to outcome n which causes the total weight to go from m to the next power of two, which we do by setting $depth = \lceil \log_2 m \rceil$ and making $a_n = 2^{depth} - m$. We then add leaves and label them so that if we sum the probabilities of all leaves labelled i, the total is equal to $a_i/2^{depth}$. The tree is represented by a table, which records, for each level c, the number of leaves $h[c]$ at that level and their labels $lab[c]$; by convention the $h[c]$ leaves occupy positions $0, \ldots, h[c]-1$ and the internal nodes follow at positions $\geq h[c]$. To decide which labels are at level c, the preprocessing step reads the binary expansions of the weights a_i and adds a leaf for each set bit $2^{depth-c}$ of a_i.

As a running example, take the weights $(a_0, a_1, a_2) = (7, 4, 8)$, so $m = 19$. Padding to the next power of two adds a reject outcome r of weight $a_3 = 2^{\lceil \log_2 19 \rceil} - 19 = 32 - 19 = 13$, giving a total of $2^5 = 32$. Reading the binary expansions $7 = (00111)_2$, $4 = (00100)_2$, $8 = (01000)_2$, and $13 = (01101)_2$, outcome i receives one leaf at depth d for each set bit 2^{5-d} of a_i; a leaf at depth d is reached with probability 2^{-d}, so the leaves of outcome i contribute exactly $a_i/32$. For instance, label 0 has leaves at depths 3, 4, 5, contributing $\frac{4+2+1}{32} = \frac{7}{32}$. Figure 4 shows the resulting tree and the stored table (h, lab), which records, for each level c, the number of leaves $h[c]$ and their labels $lab[c]$ in ascending order.

Similar to the alias method, the proof of the sampler is *parametric* over the preprocessed data structure it is handed: the table is represented by the executable `FldrTable`, and the sampler's only assumption about it is a well-formedness predicate wf. This predicate collects the pure properties the table must satisfy for a faithful encoding of the DDG: the **Vec** fields have the expected lengths, and the ghost view t@ satisfies valid_ddg. Just like valid_alias , valid_ddg is a pure predicate on the ghost view of the table (whose **Vec** fields become mathematical spec_fns), but its definition is considerably more involved, so we elide it here.



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

Fig. 4. The DDG tree (left) and the stored table (h, lab) (right) for the weights $(7, 4, 8)$, the reject label r = 3. Dashed arrows show that reaching a reject leaf restarts the walk at the root.

```
pub struct FldrTable {
  pub n: u64,   // # of labels
  pub m: u64,   // total weight
  pub levels: u64, // ceil(log2 m)
  pub weights: Vec<u64>, // a_0..a_{n-1}
  pub h: Vec<u64>,    // # leaves per level
  pub lab: Vec<Vec<u64>>, // labels in each level
}
```

```
pub open spec fn wf(t: FldrTable) -> bool {
 &&& valid_ddg(t@)
 &&& t.h@.len() == t.levels + 1
 &&& t.lab@.len() == t.levels + 1
 &&& forall|c| 0 <= c <= t.levels
    ==> t.lab@[c]@.len() == t.h@[c]
}
```

```
pub fn fldr_preprocess(
  weights: Vec<u64>, m: u64, levels: u64,
) -> (tab: FldrTable)
  requires
    1 <= levels <= 62,
    pow2(levels as nat) <= usize::MAX as nat,
    1 <= m as nat <= pow2(levels as nat),
    /* ...more requires omitted... */
  ensures
    wf(tab),
    tab.n as nat == weights@.len(),
    tab.m == m,
    tab.levels == levels,
    tab.weights@ == weights@,
{
  /* body and proof omitted */
}
```

```
pub fn sample_fldr(
  tab: &FldrTable,
  Ghost(e): Ghost<spec_fn(nat) -> real>,
  Tracked(in_credit):Tracked<ErrorCreditResource>,
  Ghost(eps): Ghost<real>,
) -> ((val, out_credit):
   (u64, Tracked<ErrorCreditResource>))
  requires
    wf(tab),
    forall |x: nat| e(x) >= 0real,
    eps >= fldr_exp(tab@, e),
    in_credit@ =~= Value { car: eps },
  ensures
    out_credit@@ =~= Value { car: e(val as nat) }
{
  /* body and proof omitted */
}
```

The sampling routine takes such a table as input, and the distribution of the samples it returns depends on that table. Again, the EPT specification is therefore stated in terms of the distribution encoded by the table, where we use the helper function `fldr_exp` to compute the expected value of the credit allocation function `e` based on this table. The verification of the sampler is very similar to that of the fast dice roller, except that the credit allocation now follows the tree's level/position structure rather than the doubling window. The credit allocation and the sampler proof are in Appendix G.

6 Extending VerusBelt for Probability

The encoding of error credits in Alerus described in §3 only involves adding two trusted axioms, which are close in formulation to the rules in Eris. Nevertheless, one might worry about potential

unsoundness because the language considered in Eris is quite different from Rust, and some of Verus's other features have no analogue in Eris. To address this and gain additional confidence in our encoding, this section adapts VerusBelt [17], which was developed to justify the soundness of some of Verus's features. Specifically, VerusBelt provides a semantic foundation for Verus's proof-oriented extensions to the Rust type system. It provides justification for a number of axioms in Verus, such as those for `PCell` and `PPtr`. Just like the axioms that we add in Alerus to connect error credits to primitive sampling functions, these connect the tracked permissions for pointers to the actual functions that modify them. Thus, by adapting the model of VerusBelt, we can similarly justify these new axioms for error credits.

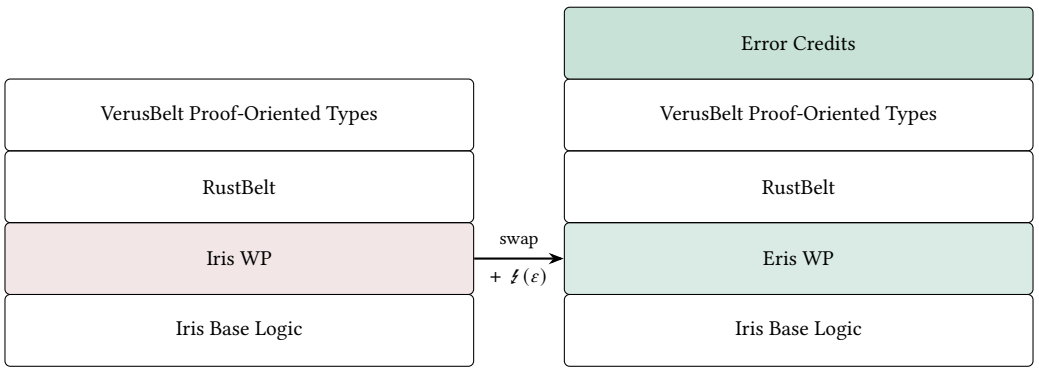


Fig. 5. The VerusBelt Proof-Oriented type proofs are written against a generic Iris weakest precondition, so porting VerusBelt to a probabilistic model involves swapping the definitions from Iris's `wp` to that of Eris. On top, we additionally define the types for the error credit resources. By ensuring that the Eris `wp` exposes the same required interface as the Iris version, we can preserve most of the existing proofs from VerusBelt.

Figure 5 illustrates how VerusBelt is constructed in layers, and how our AlerusBelt model adapts them. At the base of VerusBelt is the Iris base logic: this is the foundational assertion logic with separation logic connectives that Iris provides. On top of this, Iris implements a language-generic weakest precondition assertion `wp`. This generic weakest precondition is instantiated for λ_{Rust}, the model of a core subset of Rust used in RustBelt. Above this, a logical-relations model of Rust's type system is constructed, in which types are interpreted as separation logic assertions in this instantiation of the weakest precondition. Finally, VerusBelt itself extends this logical relations model with support for `ensures` and `requires` clauses, and validates the axioms for various primitives.

RustBelt and VerusBelt are substantial and complex Rocq developments. Thus, to make this adaptation feasible, AlerusBelt ports the development in a way that largely avoids modifying these layers. As shown in the figure, we replace the Iris weakest precondition with the Eris weakest precondition in the lower layers. Since Eris is related to Iris, it is possible to do this replacement in a way that preserves the interface that RustBelt and VerusBelt need without disruption. Then, on top of VerusBelt's existing constructions, we additionally model the error-credit resource and axioms we use, via a simple translation to the underlying error-credit rules that Eris provides.

However, pulling off the swap of Iris for Eris in this construction requires a few changes, since the formulation of Eris developed by Aguirre et al. [1] has certain mismatches with the standard Iris weakest precondition. The rest of this section describes these issues and how we resolved them. Several of these proof porting efforts were assisted by Opus 4.8.

Later Credits. VerusBelt uses an Iris feature called later credits [43], which helps manage the *later modality* that occurs in Iris in order to soundly incorporate various forms of impredicativity. However, Eris did not support later credits, as doing so posed an obstacle in its soundness proof. The issue is that, under the hood, Iris's and Eris's weakest preconditions are defined in terms of various sequences of modalities that are used to model features like updating ghost state, accessing Iris's impredicative invariants, and spending/redistributing error credits. The soundness proof for Eris relies on various *commutative laws* that allow for swapping the order of these modalities with other connectives, in particular universal quantification. Once later credits are included, however, these commutative laws are lost. As a result, the existing Eris soundness proof does not work.

Recently, however, Crawford [12] proposed an alternative formulation of several modalities in Iris and restructured Iris's soundness proof in a way that retains the key commutative laws, which was subsequently reworked by Krebbers [25]. By using their approach with these new modalities, we were able to add later credits into Eris and still derive soundness.

Prophecy Variables. Since the work of Jung et al. [24], Iris has supported *prophecy variables*, a mechanism that allows for "predicting" in the course of a proof what the future outcomes of various operations will be. VerusBelt uses this to model an extension to Verus that supports more flexible reasoning about mutable references, following an approach introduced by Matsushita et al. [34]. However, Iris's prophecy variables are known to be unsound when combined with Eris's error credits [44]. Thus, we remove all features of VerusBelt that depend upon prophecy variables. Fortunately, the only use is to model this recently implemented extension for reasoning about returning mutable references, and none of our example proofs make use of this feature.

Limitations. Besides the removal of prophecy variables, AlerusBelt has two main limitations. First, the original Eris does not support concurrency, and so we have additionally removed concurrency from λ_{Rust} in porting the model. However, none of our case studies involve concurrency. A subsequent extension to Eris developed by Li et al. [30] does add concurrency support, and it would be interesting to port AlerusBelt to this logic to recover concurrency.

A second limitation arises from the fact that VerusBelt uses only a partial-correctness variant of Iris's weakest precondition and does not model Verus's `decreases` clauses for termination. AlerusBelt thus similarly uses the partial-correctness version of Eris. Thus, while our examples use total Eris's approach to proving almost-sure termination by credit amplification, the soundness of this aspect of our encoding is not captured in AlerusBelt.

Nevertheless, this model does provide some assurance that Alerus's encoding of error credits is compatible with Verus's other features. This fact is *a priori* non-obvious, as the example of prophecies for mutable borrows and the role of later credits in VerusBelt demonstrate.

7 Related Work

Semi-Automated Verifiers for Probabilistic Programs. Caesar [40] is a deductive verifier for probabilistic programs that uses the HeyVL [41] intermediate verification language. In particular, in the latter, verification conditions are quantitative properties about expected values. HeyVL and hence Caesar can check properties that are not directly captured by specifications with Alerus, such as bounds on expected values of running times and properties like *positive* almost-sure termination. On the other hand, supporting quantitative verification conditions requires a different design and implementation of the verification stack, as compared to other automated deductive verifiers. Currently, Caesar targets a front-end language that is simpler than Rust and lacks many of the challenging language features Rust has.

Cohen [11] develops an approach to reasoning about probabilistic behaviors in deductive verifiers by encoding probabilistic quantities as ghost values that are updated in an expectation-preserving

way, much as Eris updates error credits. He uses these to state probabilistic invariants over a system's behavior. However, to the best of our knowledge, his encoding does not have an analogue of the thin-air credit rule, which we use to give modular proofs of almost-sure termination. In addition, his ghost values are not embedded in a separation logic, and so do not support the kind of splitting via separating conjunction that error credits enjoy.

Zaiser et al. [49] develop a library for probabilistic verification in Dafny. The library uses a monadic style in which programs consume streams of random bits, following an approach pioneered by Hurd [22]. They apply this approach to verify the negative exponential Bernoulli sampler discussed in §5.1.1. In contrast, Alerus can reason directly about probabilistic programs that are not written in monadic style.

Arnold [3] applies Prusti [4] to verify properties in OpenDP; however, that work focuses on non-probabilistic functional correctness properties of parts of the library, in contrast to the probabilistic samplers we have verified.

Prior Verifications of Similar Case Studies. The family of discrete Gaussian samplers described in §5.1 has also been verified in SampCert [13]. SampCert is written in a probabilistic-monadic DSL, called SLang, embedded in Lean. Sampler correctness is established by showing that the monadic denotation of the program is the intended distribution. Executable samplers are produced by extracting or compiling this DSL code. We instead reason about the executable Rust code syntactically through a program logic, using error credits as a separation logic resource. This also lets us verify interaction with Rust code that would fall outside a monadic DSL like SLang.

Zilken et al. [52] present a Hoare logic that they use to verify FDR and FLDR with a pencil-and-paper proof. Their logic allows for stating *distributional invariants*, which are invariants over the distribution of values stored in a variable. This leads to elegant loop invariants and proofs for FDR and FLDR. The lifting-based approach underlying Eris and hence Alerus does not support such distributional invariants. Nevertheless, one can interpret the credit allocation function used in our proofs in §5.3 as describing how the expected value transforms across iterations, which *does* indirectly encode the distribution. Moreover, the lifting-based approach allows us to reuse Verus's existing support for reasoning about the array-based representation of the FLDR tree.

8 Results and Future Work

We presented Alerus, a lightweight extension of Verus with probabilistic error credits. We demonstrated Alerus by verifying Rust samplers for the discrete Laplace and discrete Gaussian, the alias method, and the fast loaded dice roller. Alerus is able to take advantage of Verus's existing SMT-backed automation and reasoning features, with soundness justified through an Eris-based adaptation of VerusBelt.

For future work, it would be interesting to find a way to extend the soundness model to support prophecy variables. While full prophecy variables are not compatible with Eris, Verus only requires a limited form of prophecies. Verus imposes restrictions on prophecies to prevent inconsistencies with other Verus features, and these restrictions might make them compatible with error credits. Additionally, Alerus could be extended to reason about concurrent systems [30] or to support relational properties of probabilistic programs, such as differential privacy [18, 50].

Acknowledgments

We thank the Verus community for their support, especially Baltasar Dinis, Travis Hance, Chris Hawblitzel, Bryan Parno, and Daniel Schoepe. We also thank Alejandro Aguirre, Markus de Medeiros, Lars Birkedal, Simon Oddershede Gregersen, Philipp Haselwarter, Kwing Hei Li, and Puming Liu for their helpful discussions. The first author especially thanks Markus for his help on demystifying

the Fast Geometric Exponential sampler. This work was supported in part by the National Science Foundation under Grant No. 2338317. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of these funding agencies.

References

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

[2] Andrew W. Appel. 2011. Verified Software Toolchain - (Invited Talk). In *ESOP (Lecture Notes in Computer Science, Vol. 6602)*. Springer, 1–17.

[3] Till Arnold. 2024. *Automated Verification of a Rust Differential Privacy Library*. Master's Thesis. ETH Zürich, Department of Computer Science. https://ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Education/Theses/Till_Arnold_MA_Report.pdf Advisors: Jonáš Fiala, Anouk Paradis, Prof. Dr. Peter Müller.

[4] Vytautas Astrauskas, Aurel Bílý, Jonáš Fiala, Zachary Grannan, Christoph Matheja, Peter Müller, Federico Poli, and Alexander J. Summers. 2022. The Prusti Project: Formal Verification for Rust. In *NFM (Lecture Notes in Computer Science, Vol. 13260)*. Springer, 88–108.

[5] Gilles Barthe, Thomas Espitau, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2018. An Assertion-Based Program Logic for Probabilistic Programs. In *Programming Languages and Systems*, Amal Ahmed (Ed.). Springer International Publishing, Cham, 117–144.

[6] Gilles Barthe, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2016. A Program Logic for Union Bounds. In *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, 2016, Rome, Italy (LIPIcs, Vol. 55)*, Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Rome, Italy, 107:1–107:15. https://doi.org/10.4230/LIPICS.ICALP.2016.107

[7] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. 2009. Formal certification of code-based cryptographic proofs. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Savannah, GA, USA) *(POPL '09)*. Association for Computing Machinery, New York, NY, USA, 90–101. https://doi.org/10.1145/1480881.1480894

[8] Gilles Barthe, Justin Hsu, and Kevin Liao. 2020. A probabilistic separation logic. *Proc. ACM Program. Lang.* 4, POPL (2020), 55:1–55:30.

[9] Clément L. Canonne, Gautam Kamath, and Thomas Steinke. 2020. The Discrete Gaussian for Differential Privacy. In *Proceedings of the 34th International Conference on Neural Information Processing Systems (NIPS '20)*. Curran Associates Inc., Red Hook, NY, USA, 15676–15688.

[10] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. 2019. Verifying concurrent, crash-safe systems with Perennial. In *SOSP*. ACM, 243–258.

[11] Ernie Cohen. 2017. Reducing probabilistic choice to nondeterministic choice. Talk at the Workshop on Probabilistic Programming Semantics (PPS 2017), Paris, France.

[12] Freja Marott Crawford. 2026. *Adequacy with Later Credits in the Iris Logic.* MSc project report. Department of Computer Science, Aarhus University. https://iris-project.org/pdfs/2026-msc-project-crawford.pdf.

[13] Markus de Medeiros, Muhammad Naveed, Tancrède Lepoint, Temesghen Kahsai, Tristan Ravitch, Stefan Zetzsche, Anjali Joshi, Joseph Tassarotti, Aws Albarghouthi, and Jean-Baptiste Tristan. 2025. Verified Foundations for Differential Privacy. *Proc. ACM Program. Lang.* 9, PLDI (2025), 1094–1118. https://doi.org/10.1145/3729294

[14] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.

[15] 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

[16] Travis Hance. 2024. *Verifying Concurrent Systems Code.* Ph.D. thesis. Carnegie Mellon University, Pittsburgh, PA, USA. https://www.andrew.cmu.edu/user/bparno/papers/hance_thesis.pdf

[17] Travis Hance, Laila Elbeheiry, Yusuke Matsushita, and Derek Dreyer. 2026. VerusBelt: A Semantic Foundation for Verus's Proof-Oriented Extensions to the Rust Type System. *Proc. ACM Program. Lang.* 10, PLDI, Article 247 (June 2026), 25 pages. https://doi.org/10.1145/3808325

[18] Philipp G. Haselwarter, Alejandro Aguirre, Simon Oddershede Gregersen, Kwing Hei Li, Joseph Tassarotti, and Lars Birkedal. 2026. Modular Verification of Differential Privacy in Probabilistic Higher-Order Separation Logic. In

Proceedings of the 47th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '26). Association for Computing Machinery, New York, NY, USA. https://simongregersen.com/papers/2026-clutchDP.pdf To appear.

[19] Philipp G. Haselwarter, Kwing Hei Li, Alejandro Aguirre, Simon Oddershede Gregersen, Joseph Tassarotti, and Lars Birkedal. 2025. Approximate Relational Reasoning for Higher-Order Probabilistic Programs. *Proc. ACM Program. Lang.* 9, POPL (2025), 1196–1226.

[20] Philipp G. Haselwarter, Kwing Hei Li, Markus de Medeiros, Simon Oddershede Gregersen, Alejandro Aguirre, Joseph Tassarotti, and Lars Birkedal. 2024. Tachis: Higher-Order Separation Logic with Credits for Expected Costs. *Proc. ACM Program. Lang.* 8, OOPSLA2 (2024), 1189–1218.

[21] Johannes Hostert, Zichen Zhang, Puming Liu, Simon Oddershede Gregersen, Ralf Jung, and Joseph Tassarotti. 2026. Completeness of Iris-Based Program Logics. *Proc. ACM Program. Lang.* ICFP (2026). https://simongregersen.com/papers/2026-completeness.pdf To appear.

[22] Joe Hurd. 2002. *Formal Verification of Probabilistic Algorithms*. Ph.D. thesis. University of Cambridge. https://doi.org/10.48456/tr-566

[23] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.* 2, POPL, Article 66 (Dec. 2017), 34 pages. https://doi.org/10.1145/3158154

[24] Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. 2020. The future is ours: prophecy variables in separation logic. *Proc. ACM Program. Lang.* 4, POPL (2020), 45:1–45:32.

[25] Robbert Krebbers. [n. d.]. Improve later credits (alternative). https://gitlab.mpi-sws.org/iris/iris/-/merge_requests/1217. Iris merge request !1217. Accessed 2026-07-06.

[26] Andrea Lattuada, Travis Hance, Jay Bosamiya, Matthias Brun, Chanhee Cho, Hayley LeBlanc, Pranav Srinivasan, Reto Achermann, Tej Chajed, Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Oded Padon, and Bryan Parno. 2024. Verus: A Practical Foundation for Systems Verification. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX, USA) *(SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 438–454. https://doi.org/10.1145/3694715.3695952

[27] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023. Verus: Verifying Rust Programs using Linear Ghost Types. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 85 (April 2023), 30 pages. https://doi.org/10.1145/3586037

[28] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Proceedings of the Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*. Springer-Verlag, 348–370.

[29] 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 (June 2023), 24 pages. https://doi.org/10.1145/3591226

[30] Kwing Hei Li, Alejandro Aguirre, Simon Oddershede Gregersen, Philipp G. Haselwarter, Joseph Tassarotti, and Lars Birkedal. 2025. Modular Reasoning about Error Bounds for Concurrent Probabilistic Programs. *Proc. ACM Program. Lang.* 9, ICFP, Article 245 (Aug. 2025), 30 pages. https://doi.org/10.1145/3747514

[31] Janine Lohse, Tim Rohde, Jimmy Xin, Niklas Mück, Iona Kuhn, Derek Dreyer, Deepak Garg, and Emanuele D'Osualdo. 2026. First Steps Towards Probabilistic Iris: Harmonizing Independence, Conditioning, and Dynamic Heap Allocation. *CoRR* abs/2605.13765 (2026).

[32] Jérémie O. Lumbroso. 2013. Optimal Discrete Uniform Generation from Coin Flips, and Applications. *CoRR* abs/1304.1916 (2013). arXiv:1304.1916 http://arxiv.org/abs/1304.1916

[33] Virgil Marionneau, Félix Sassus Bourda, Alejandro Aguirre, and Lars Birkedal. 2026. Modular Specifications and Implementations of Random Samplers in Higher-Order Separation Logic. In *Proceedings of the 15th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Rennes, France) *(CPP '26)*. Association for Computing Machinery, New York, NY, USA, 368–382. https://doi.org/10.1145/3779031.3779109

[34] Yusuke Matsushita, Xavier Denis, Jacques-Henri Jourdan, and Derek Dreyer. 2022. RustHornBelt: a semantic foundation for functional verification of Rust programs with unsafe code. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 841–856. https://doi.org/10.1145/3519939.3523704

[35] Annabelle McIver and Carroll Morgan. 2005. *Abstraction, Refinement and Proof for Probabilistic Systems*. Springer. https://doi.org/10.1007/B138392

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

[37] 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

[38] 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

[39] Feras Saad, Cameron Freer, Martin Rinard, and Vikash Mansinghka. 2020. The Fast Loaded Dice Roller: A Near-Optimal Exact Sampler for Discrete Probability Distributions. In *Proceedings of the Twenty Third International Conference on Artificial Intelligence and Statistics (Proceedings of Machine Learning Research, Vol. 108)*, Silvia Chiappa and Roberto Calandra (Eds.). PMLR, 1036–1046. https://proceedings.mlr.press/v108/saad20a.html

[40] Philipp Schröer, Kevin Batz, Umut Yigit Dural, Darion Haase, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. 2026. Caesar: A Deductive Verifier for Probabilistic Programs. *CoRR* abs/2605.15827 (2026).

[41] Philipp Schröer, Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. 2023. A Deductive Verification Infrastructure for Probabilistic Programs. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 294 (Oct. 2023), 31 pages. https://doi.org/10.1145/3622870

[42] Keith Schwarz. 2011. Darts, Dice, and Coins: Sampling from a Discrete Distribution. https://www.keithschwarz.com/darts-dice-coins/. Accessed 2026-06-16.

[43] 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.

[44] The Clutch Development Team. [n. d.]. Prophecy variables are unsound with up-to-bad reasoning. https://github.com/logsem/clutch/blob/main/theories/eris/examples/noproph.v. Rocq mechanization in the `logsem/clutch` repository, file `theories/eris/examples/noproph.v`. Accessed 2026-07-06.

[45] The OpenDP Project. 2026. OpenDP. https://github.com/opendp/opendp.

[46] Michael D. Vose. 1991. A Linear Algorithm for Generating Random Numbers with a Given Distribution. *IEEE Transactions on Software Engineering* 17, 9 (1991), 972–975. https://doi.org/10.1109/32.92917

[47] Alastair J. Walker. 1977. An Efficient Method for Generating Discrete Random Variables with General Distributions. *ACM Trans. Math. Software* 3, 3 (1977), 253–256. https://doi.org/10.1145/355744.355749

[48] Felix A. Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, João Carlos Pereira, and Peter Müller. 2021. Gobra: Modular Specification and Verification of Go Programs. In *CAV (1) (Lecture Notes in Computer Science, Vol. 12759)*. Springer, 367–379.

[49] Fabian Zaiser, Stefan Zetzsche, and Jean-Baptiste Tristan. 2024. VMC: a Dafny Library for Verified Monte Carlo Algorithms. Talk at the Dafny 2024 Workshop (co-located with POPL 2024), London, UK.

[50] Danfeng Zhang and Daniel Kifer. 2017. LightDP: towards automating differential privacy proofs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 888–901. https://doi.org/10.1145/3009837.3009884

[51] Jacob Zhong. 2024. dashu: A library set of arbitrary precision numbers implemented in Rust. https://crates.io/crates/dashu.

[52] Daniel Zilken, Kevin Batz, Joost-Pieter Katoen, and Tobias Winkler. 2026. Verifying Sampling Algorithms via Distributional Invariants. In *Formal Methods*, Augusto Sampaio and Marielle Stoelinga (Eds.). Springer Nature Switzerland, Cham, 259–278.

A Convergence of the Random-Walk Fail Probability

Recall the credit allocation for the 1-dimensional random walk,

$$\texttt{fail\_prob}(s, p) \;=\; 1 - \Pr[\text{walk from } p \text{ reaches } 0 \text{ within } s \text{ steps}],$$

which satisfies $\texttt{fail\_prob}(s, 0) = 0$, $\texttt{fail\_prob}(0, p) = 1$ for $p > 0$, and $\texttt{fail\_prob}(s, p) = \frac{1}{2}\big(\texttt{fail\_prob}(s{-}1, p{-}1) + \texttt{fail\_prob}(s{-}1, p{+}1)\big)$. To justify picking a sufficient fuel we must show

$$\forall \delta > 0.\ \exists s.\ \texttt{fail\_prob}(s, \texttt{pos}) < \delta.$$

We prove this by (i) showing $\texttt{fail\_prob}(\cdot, p)$ is non-increasing and bounded in $[0, 1]$, so by monotone convergence it has a limit $\texttt{fail\_limit}(p)$; (ii) taking limits in the recurrence to obtain the equation

$$\texttt{fail\_limit}(p) \;=\; \tfrac{1}{2}\big(\texttt{fail\_limit}(p{-}1) + \texttt{fail\_limit}(p{+}1)\big), \qquad \texttt{fail\_limit}(0) = 0,$$

whose general solution forces

$$\texttt{fail\_limit}(p) \;=\; p \cdot \texttt{fail\_limit}(1);$$

and (iii) ruling out $\texttt{fail\_limit}(1) > 0$, which would give $\texttt{fail\_limit}(p) > 1$ for some p, contradicting the bound. Hence $\texttt{fail\_limit}(p) = 0$ for every p, so $\texttt{fail\_prob}(\cdot, \texttt{pos})$ converges to 0 and drops below any $\delta > 0$ at some finite s.

B Invariant Preservation for the Negative-Exponential Bernoulli Sampler

Recall the loop invariant for `sample_bernoulli_exp1` from §5.1.1: on entry to iteration k the held credit ε_k satisfies $\varepsilon_k \geq p_k\, \mathcal{E}(\texttt{true}) + (1 - p_k)\, \mathcal{E}(\texttt{false})$, where the return probabilities p_k obey $p_k = \frac{x}{k}\, p_{k+1} + (1 - \frac{x}{k})\, [k \text{ odd}]$. On the `true` branch of iteration k the credit carried into the next iteration is $\varepsilon_{k+1} = \frac{k}{x}\, \varepsilon_k - (\frac{k}{x} - 1)\, \mathcal{E}([k \text{ odd}])$, and we check that the invariant still holds at step $k + 1$:

$$\begin{aligned}
\varepsilon_{k+1} &= \tfrac{k}{x}\, \varepsilon_k - \big(\tfrac{k}{x} - 1\big)\mathcal{E}([k \text{ odd}]) \\
&\geq \tfrac{k}{x}\big(p_k\, \mathcal{E}(\texttt{true}) + (1 - p_k)\, \mathcal{E}(\texttt{false})\big) - \big(\tfrac{k}{x} - 1\big)\mathcal{E}([k \text{ odd}]) \\
&= p_{k+1}\, \mathcal{E}(\texttt{true}) + (1 - p_{k+1})\, \mathcal{E}(\texttt{false}),
\end{aligned}$$

using $\mathcal{E}([k \text{ odd}]) = [k \text{ odd}]\, \mathcal{E}(\texttt{true}) + (1 - [k \text{ odd}])\, \mathcal{E}(\texttt{false})$ and the recurrence rearranged as $\frac{k}{x}\, p_k - \big(\frac{k}{x} - 1\big)[k \text{ odd}] = p_{k+1}$. The invariant is thus preserved, and since $p_{k+1} \in [0, 1]$ and $\mathcal{E} \geq 0$ the resulting ε_{k+1} is hence a valid credit.

C The Taylor-Tail Axiom for e^{-x}

Verus's SMT backend has no theory of Euler's number e, nor does it have completeness of reals. Alerus exposes $e^{(-)}$ as an uninterpreted function and axiomatizes only the facts it needs. The single nontrivial axiom is the alternating-series bound on the Taylor partial sums of e^{-x}, which the negative-exponential sampler of §5.1.1 uses to show its conditional probabilities p_k lie in $[0, 1]$. We discharge it separately in Lean, so the Verus-level development depends only on this small, clearly delimited interface.

```
pub uninterp spec fn exp(x: real) -> real;          // e^x, uninterpreted

pub open spec fn exp_taylor_term(x: real, k: nat) -> real {
  pow(-x, k) / factorial(k)                 // (-x)^k / k!
}
pub open spec fn exp_taylor_seq(x: real) -> spec_fn(nat) -> real {
  |k: nat| exp_taylor_term(x, k)
}

// T_n = partial_sum(exp_taylor_seq(x), n) = sum_{k<n} (-x)^k / k!
```

```
#[verifier::external_body]
pub proof fn axiom_exp_taylor_bounds(x: real, n: nat)
  requires 0real < x <= 1real, n >= 1,
  ensures
    0real <= partial_sum(exp_taylor_seq(x), n),      // T_n in [0,1]
    partial_sum(exp_taylor_seq(x), n) <= 1real,
    n % 2 == 0 ==> partial_sum(exp_taylor_seq(x), n) <= exp(-x),  // even: underestimate
    n % 2 == 1 ==> exp(-x) <= partial_sum(exp_taylor_seq(x), n),  // odd: overestimate
{}
```

D The Fast Geometric Sampler

For $x = n/d$ in lowest terms, `sample_geometric_exp_fast` draws u from the exponential-rejection sampler, $\Pr[u] = e^{-u/d}/N$ with $N = \sum_{u=0}^{d-1} e^{-u/d}$, and an independent $v \sim \mathrm{Geom}(1-e^{-1})$ with $\Pr[v] = e^{-v}(1-e^{-1})$, and returns $\lfloor (u + d\,v)/n \rfloor$. Working backwards from EPT, the slow geometric sampler is handed $g(u,v) = \mathcal{E}\big((u+d\,v)/n\big)$, forcing the rejection sampler's allocation

$$f(u) \;=\; \mathbb{E}_{v\sim\mathrm{Geom}(1-e^{-1})}[\,g(u,v)\,] \;=\; \sum_{v=0}^{\infty} e^{-v}(1-e^{-1})\,\mathcal{E}\big((u+d\,v)/n\big).$$

The precondition $\varepsilon = \sum_{r=0}^{\infty}(e^{-n/d})^r(1-e^{-n/d})\,\mathcal{E}(r)$ then covers the rejection sampler's credit, using two Euclidean-division bijections, $k := u + d\,v$ over $\{0,\dots,d-1\}\times\mathbb{N}$ and $k = n\,r + i$ over $\mathbb{N}\times\{0,\dots,n-1\}$:

$$\begin{aligned}
\mathbb{E}_{u\sim\mathrm{rejection}}[\,f(u)\,] &= \tfrac{1}{N}\sum_{u=0}^{d-1} e^{-u/d}\,f(u) \\
&= \tfrac{1-e^{-1}}{N}\sum_{u=0}^{d-1}\sum_{v\in\mathbb{N}} e^{-(u+dv)/d}\,\mathcal{E}\big((u+d\,v)/n\big) && \text{(unfold } f\text{)} \\
&= \tfrac{1-e^{-1}}{N}\sum_{k\in\mathbb{N}} e^{-k/d}\,\mathcal{E}(k/n) && (k = u + d\,v) \\
&= \tfrac{1-e^{-1}}{N}\sum_{r\in\mathbb{N}}\sum_{i<n} e^{-(nr+i)/d}\,\mathcal{E}(r) && (k = n\,r + i) \\
&= \tfrac{1-e^{-1}}{N}\,\tfrac{1-e^{-n/d}}{1-e^{-1/d}}\sum_{r\in\mathbb{N}}(e^{-n/d})^r\,\mathcal{E}(r) \\
&= \sum_{r\in\mathbb{N}}(e^{-n/d})^r(1-e^{-n/d})\,\mathcal{E}(r) && \big(N = \tfrac{1-e^{-1}}{1-e^{-1/d}}\big).
\end{aligned}$$

E Credit Split for the Discrete Laplace Sampler

The discrete Laplace sampler of §5.1.3 flips a fair sign and then, inside the chosen branch, draws the magnitude k with `sample_geometric_exp_fast`, a $\mathrm{Geom}(1-p)$ sampler (§5.1.2). We hand that magnitude draw the allocation $k \mapsto \mathcal{E}(k)$ in the positive branch and $k \mapsto \mathcal{E}_{-}(k)$ in the negative branch, where $\mathcal{E}_{-}(k) = \mathcal{E}(-k)$ for $k \geq 1$ and $\mathcal{E}_{-}(0) = \varepsilon$. By that sampler's EPT precondition, the credit each branch must supply is the $\mathrm{Geom}(1-p)$-average of its allocation, so backward execution through the sign flip yields the two branch budgets

$$\varepsilon_{+} = \mathbb{E}_{k\sim\mathrm{Geom}(1-p)}[\,\mathcal{E}(k)\,] = \sum_{k} p^k(1-p)\,\mathcal{E}(k), \qquad \varepsilon_{-} = \sum_{k} p^k(1-p)\,\mathcal{E}_{-}(k).$$

The sign flip's own EPT precondition then requires its average $\frac{1}{2}(\varepsilon_+ + \varepsilon_-)$ to fit within ε. Unfolding the two budgets and splitting off the $k = 0$ term of each,

$$\begin{aligned}\varepsilon_+ + \varepsilon_- &= (1-p)\,\varepsilon + (1-p)\,\mathcal{E}(0) + \sum_{k\geq 1} p^k(1-p)\big(\mathcal{E}(k) + \mathcal{E}(-k)\big)\\ &= (1-p)\,\varepsilon + (1+p)\Big(\mu_L(0)\,\mathcal{E}(0) + \sum_{k\geq 1}\big(\mu_L(k)\,\mathcal{E}(k) + \mu_L(-k)\,\mathcal{E}(-k)\big)\Big)\\ &= (1-p)\,\varepsilon + (1+p)\sum_{x\in\mathbb{Z}}\mu_L(x)\,\mathcal{E}(x) \;\leq\; (1-p)\,\varepsilon + (1+p)\,\varepsilon \;=\; 2\,\varepsilon,\end{aligned}$$

so $\frac{1}{2}(\varepsilon_+ + \varepsilon_-) \leq \varepsilon$ and the sign flip's precondition is met.

F The Discrete Gaussian Sampler

The discrete Gaussian $\mathcal{N}_{\mathbb{Z}}(0,\sigma^2)$ has $\mathrm{pmf}(x) = e^{-x^2/2\sigma^2}/Z$ with $Z = \sum_{y\in\mathbb{Z}} e^{-y^2/2\sigma^2}$. As noted in §5.1.4, it is sampled by rejection against a discrete-Laplace proposal: with $t = \lfloor\sigma\rfloor + 1$, we draw $y \sim \mathrm{Lap}(0,t)$ and accept with probability $C(y) = e^{-\mathrm{bias}(y)}$, where $\mathrm{bias}(y) = (|y| - \sigma^2/t)^2/(2\sigma^2)$.

```
pub fn sample_discrete_gaussian(sigma: &RBig) -> IBig {
  let t: RBig = (sigma.floor() + 1).into();
  let sigma2 = sigma * sigma;
  loop {
    let y = sample_discrete_laplace(&t);
    let bias = (y.abs() - &sigma2 / &t).square() / (2 * &sigma2);
    if sample_bernoulli_exp(&bias) {
      return y;
    }
  }
}
```

Writing $p = e^{-1/t}$, the discrete-Laplace proposal has pmf $\mu_L(y) = \frac{1-p}{1+p}\,e^{-|y|/t}$. The key identity is that the proposal weight times the acceptance probability is a multiple of the target: since $|y|/t + \mathrm{bias}(y) = y^2/2\sigma^2 + \sigma^2/2t^2$,

$$\mu_L(y)\,C(y) = \tfrac{1-p}{1+p}\,e^{-|y|/t}\,e^{-\mathrm{bias}(y)} = \kappa\,e^{-y^2/2\sigma^2} = a\,\mathrm{pmf}(y), \qquad \kappa = \tfrac{1-p}{1+p}\,e^{-\sigma^2/2t^2}, \quad a = \kappa Z,$$

Summing over y, one iteration accepts with probability $a = \sum_y \mu_L(y)\,C(y) = \kappa Z$, and conditioned on accepting it returns y with probability $\kappa\,e^{-y^2/2\sigma^2}/a = \mathrm{pmf}(y)$.

As in all rejection samplers, the loop maintains $\varepsilon \geq \sum_{x\in\mathbb{Z}} \mathrm{pmf}(x)\,\mathcal{E}(x)$. The acceptance flip $\mathrm{Bern}(C(y))$ pays out $\mathcal{E}(y)$ on `true` (accept and return y) and carries ε back on `false` (reject and restart), so backward execution through the $\mathrm{Lap}(0,t)$ proposal draw forces the allocation $g(y) = C(y)\,\mathcal{E}(y) + (1-C(y))\,\varepsilon$. Discharging its EPT precondition, $\sum_y \mu_L(y)\,g(y) \leq \varepsilon$, unfolds via the factorization $\mu_L(y)\,C(y) = a\,\mathrm{pmf}(y)$ above as

$$\begin{aligned}\sum_y \mu_L(y)\,g(y) &= \sum_y \mu_L(y)\,C(y)\,\mathcal{E}(y) + \varepsilon\sum_y \mu_L(y)\big(1-C(y)\big)\\ &= \sum_y a\,\mathrm{pmf}(y)\,\mathcal{E}(y) + \varepsilon\Big(\textstyle\sum_y \mu_L(y) - \sum_y \mu_L(y)\,C(y)\Big) \qquad \big(\mu_L(y)\,C(y) = a\,\mathrm{pmf}(y)\big)\\ &= a\sum_{x\in\mathbb{Z}} \mathrm{pmf}(x)\,\mathcal{E}(x) + (1-a)\,\varepsilon \qquad \big(\textstyle\sum_y \mu_L(y) = 1,\ \sum_y \mu_L(y)\,C(y) = a\big)\\ &\leq a\,\varepsilon + (1-a)\,\varepsilon \;=\; \varepsilon,\end{aligned}$$

the inequality by the precondition $\sum_x \mathrm{pmf}(x)\,\mathcal{E}(x) \le \varepsilon$. So the held credit ε covers the proposal draw: on acceptance the loop returns y holding $\mathcal{E}(y)$, and on rejection it carries ε back to restart, exactly as in the discrete-Laplace loop.

G Mathematical Bound on FDR Credit Allocation

We prove the bound by summing all possible outcomes for $c \in [0, v)$: $S(v,k) = \sum_{c<v} \mathrm{fdr}_f(v,c,k)$ and establishing the *uniform-in-v* bound:

$$\forall v. \quad S(v,k) \;\le\; v \cdot \mathrm{avg}(n,\mathcal{E}).$$

We prove this by induction on fuel k. Unfolding fdr_f and reindexing the resulting pair sum,

$$S(v,k) \;=\; \sum_{c<v} \tfrac{1}{2}\big(\mathrm{fdr}_h(2v,\,2c,\,k{-}1) + \mathrm{fdr}_h(2v,\,2c{+}1,\,k{-}1)\big) \;=\; \tfrac{1}{2}\sum_{c<2v} \mathrm{fdr}_h(2v,\,c,\,k{-}1),$$

and, when $2v \ge n$, a threshold split at $c = n$ separates the accept terms from the reject terms,

$$\begin{aligned}
S(v,k) &= \tfrac{1}{2}\sum_{c<2v} \mathrm{fdr}_h(2v,\,c,\,k{-}1) \\
&= \tfrac{1}{2}\Big(\underbrace{\sum_{c<n} \mathcal{E}(c)}_{\text{accept}} \;+\; \underbrace{\sum_{n\le c<2v} \mathrm{fdr}_f(2v{-}n,\,c{-}n,\,k{-}1)}_{\text{reject, restart}}\Big) \\
&= \tfrac{1}{2}\big(\textstyle\sum_{i<n} \mathcal{E}(i) + S(2v{-}n,\,k{-}1)\big) \\
&\le \tfrac{1}{2}\big(\textstyle\sum_{i<n} \mathcal{E}(i) + (2v{-}n)\cdot \mathrm{avg}(n,\mathcal{E})\big) && \text{(IH)} \\
&= \tfrac{1}{2}\big(n + (2v{-}n)\big)\cdot \mathrm{avg}(n,\mathcal{E}) \;=\; v\cdot \mathrm{avg}(n,\mathcal{E}), && \big(\textstyle\sum_{i<n} \mathcal{E}(i) = n\cdot \mathrm{avg}(n,\mathcal{E})\big)
\end{aligned}$$

where the second equality reindexes $c \mapsto c - n$ over $[n, 2v)$, and the inequality applies the inductive hypothesis at fuel $k{-}1$. The sub-threshold case $2v < n$ is similar but with no accept terms. The base case $k = 0$ is immediate, since $\mathrm{fdr}_f(\cdot, 0) = 0$ makes $S(v,0) = 0$. Finally, instantiating the uniform bound at the loop's start level $v = 1$ gives the claim: the sum $S(1,k) = \sum_{c<1} \mathrm{fdr}_f(1,c,k)$ has the single term $\mathrm{fdr}_f(1,0,k)$, so $\mathrm{fdr}_f(1,0,k) \le \mathrm{avg}(n,\mathcal{E})$.

The loaded case (FLDR). The verification uses the analogous pair of mutually recursive credit allocations, now following the DDG tree's level/position structure rather than the doubling window. Here $\mathrm{fldr}_f(c,d,k)$ is the conditional expectation $\mathbb{E}[\,\mathcal{E}(\mathit{out}) \mid (c,d)\,]$ over the next k coin flips from the node at depth c and position d, and fldr_g resolves a node after the flip:

$$\begin{aligned}
&\mathrm{fldr}_f(c,d,0) = 0 && && \text{(ran out of fuel } k\text{)} \\
&\mathrm{fldr}_f(c,d,k) = \tfrac{1}{2}\big(\mathrm{fldr}_g(c{+}1,\,2d,\,k{-}1) + \mathrm{fldr}_g(c{+}1,\,2d{+}1,\,k{-}1)\big) && && \\
&\mathrm{fldr}_g(c,d,k) = \mathcal{E}(\mathit{lab}[c][d]) && \text{if } d < h[c],\ \mathit{lab}[c][d] < n && \text{(accept)} \\
&\qquad\qquad \mathrm{fldr}_f(0,\,0,\,k) && \text{if } d < h[c],\ \mathit{lab}[c][d] = n && \text{(reject, restart)} \\
&\qquad\qquad \mathrm{fldr}_f(c,\,d{-}h[c],\,k) && \text{if } d \ge h[c] && \text{(internal, descend)}
\end{aligned}$$

Mimicking the program, fldr_g pays out $\mathcal{E}(\mathit{lab}[c][d])$ on a real-outcome leaf, restarts at the root node $(0,0)$ on a reject leaf ($\mathit{lab}[c][d] = n$), and otherwise renumbers $d \mapsto d - h[c]$ to descend into the internal nodes. The termination allocation $\mathrm{fldr\_fail}_f$ again mirrors this recursion, with the accepting payout replaced by 0 and the out-of-fuel base case set to 1.

The fast loaded dice roller bound is proved the same way: we show the claim $\mathrm{fldr}_f(0,0,k) \le T$ ($T := \sum_{i<n} \frac{a_i}{m}\,\mathcal{E}(i)$) by induction on the fuel k. Unfolding $\mathrm{fldr}_f(0,0,k)$: each leaf (c,d) is reached

with probability 2^{-c}, an accept leaf contributes $\mathcal{E}(lab[c][d])$, each reject leaf restarts at the root with the remaining fuel $k - c$:

$$\begin{aligned}
\mathrm{fldr}_f(0,0,k) &= \sum_{\text{accept}} 2^{-c}\,\mathcal{E}(lab) \;+\; \sum_{\text{reject}} 2^{-c}\,\mathrm{fldr}_f(0,0,k-c) \\
&\le \sum_{\text{accept}} 2^{-c}\,\mathcal{E}(lab) \;+\; T \sum_{\text{reject}} 2^{-c} && \text{(IH)} \\
&= \tfrac{m}{2^{depth}}\,T \;+\; \left(1 - \tfrac{m}{2^{depth}}\right) T \;=\; T, && \text{(leaf-sum identity)}
\end{aligned}$$

where $depth = \lceil \log_2 m \rceil$

The regrouping is the DDG leaf-sum identity: collecting the accept leaves by label gives $\sum_{\text{accept}} 2^{-c}\mathcal{E}(lab) = \sum_{i<n} \frac{a_i}{2^{depth}}\mathcal{E}(i) = \frac{m}{2^{depth}}\,T$, while the reject leaves carry $\sum_{\text{reject}} 2^{-c} = 1 - m/2^{depth}$. The induction is well founded because every leaf has depth ≥ 1.