Verifying Exact Samplers for Continuous Distributions with a Discrete Program Logic

Markus de Medeiros
New York University, NY, USA

Puming Liu
NYU Shanghai, China

Kwing Hei Li
Aarhus University, Denmark

Alejandro Aguirre
Aarhus University, Denmark

Lars Birkedal
Aarhus University, Denmark

Joseph Tassarotti
New York University, NY, USA

Abstract

Most implementations of sampling algorithms for continuous distributions use floating-point numbers, which introduce round-off errors and approximations. These errors can be difficult to analyze, and can cause security issues when used in algorithms for differential privacy. An alternative is to use *exact* sampling algorithms based on computable reals, which can lazily generate the digits of a continuous sample to arbitrary precision. However, these algorithms are intricate, and implementing and using them involves a combination of semantically challenging language features, such as probabilistic choice, higher-order functions, and dynamically-allocated mutable state.

In this paper we present Continuous-Eris, a higher-order separation logic for verifying the correctness of exact sampling algorithms for computable distributions. To demonstrate Continuous-Eris, we verify the correctness of computable samplers for the uniform, Gaussian, and Laplace distributions, as well as a library for exact real arithmetic for working with generated samples. All of the results in this paper have been verified in the Rocq proof assistant.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Program verification; Theory of computation → Probabilistic computation

Keywords and phrases Probabilistic Programming, Separation Logic, Formal Verification

Digital Object Identifier 10.4230/LIPIcs.LICS.2026.71

Related Version *Full Version*: `https://arxiv.org/abs/2605.13526` [16]

Supplementary Material *Software*: `https://zenodo.org/records/20114732`

Funding This work was supported in part by the National Science Foundation, grant no. 2338317, a Villum Investigator grant, no. 25804, Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation, and the European Union (ERC, CHORDS, 101096090). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them. Joseph Tassarotti is also employed by Amazon Web Services (AWS). This work does not reflect the views of AWS and is not connected to any products or services provided by AWS.

```
U _ ≜ (ref None, tape 1)
ForceNext ℓ α ≜ match ! ℓ with
                | Some (b, ℓ') ⇒ (b, ℓ')
                | None         ⇒ let b := rand α 1 in
                                 let ℓ' := ref None in
                                 ℓ ← Some (b, ℓ');;
                                 (b, ℓ')
                end
GetBits ℓ α N A ≜ if N = 0 then A else
                  let (b, ℓ') := ForceNext ℓ α in
                  GetBits ℓ' α (N − 1) (2 ∗ A + b)
```

Figure 1 A lazy exact sampler for the uniform distribution over the interval $[0, 1]$, and a client GetBits.

1 Introduction

When computing with real-valued data, floating-point numbers are commonly used. Because of their bounded precision, operations on floating-point numbers inevitably introduce round-off errors. For many applications, minor approximation errors are acceptable, and an expert who understands floating-point arithmetic can design algorithms to minimize such errors. However, for non-experts, the behaviors of floating-point arithmetic can sometimes be confusing, and in some applications, it is difficult to understand how much approximation error a computation introduces. These floating-point errors can even lead to critical bugs or security issues. For example, differentially private algorithms are often formally developed under the assumption that arithmetic is carried out exactly, and the presence of floating-point errors can lead to private information leakage [45].

One alternative approach, which avoids round-off error, is to use algorithms for *exact real arithmetic* over computable real numbers. A number of different styles of exact real arithmetic exist, but the high level idea behind these approaches is to represent a real number by a function f which can be queried to get increasingly more precise approximations of the number. Arithmetic operations on these numbers are higher-order programs, returning new functions which approximate the result of a real-valued operation in terms of approximations of its arguments. This idea has its origins in constructive mathematics and the earliest days of computer science, and has been well-studied from both a theoretical perspective [48, 21, 25, 51, 54] and an applied point of view, with several different implementations of exact reals as libraries or as built-in features of programming languages [43, 37, 10, 46, 6].

What is less well-known is that by representing real numbers as functions, it is also possible to generate exact samples from continuous probability distributions. Several algorithms exist for converting an infinite sequence of uniform Bernoulli samples into exact samples from the uniform distribution over the interval $[0, 1]$, the exponential distribution, and the Gaussian distribution, among others. Using these algorithms, it becomes possible to do Monte Carlo simulations or other calculations with continuous random samples without approximations or round-off errors.

Figure 1 shows a simple implementation of an algorithm that lazily samples an exact value from the uniform distribution over the interval $[0, 1]$, written in an ML-like language (for now, ignore the code printed in a light gray color). The algorithm stores the sampled

value as a mutable linked list of bits which represent the binary digits of the number that have been sampled so far. The sampling function U simply creates an empty list, encoded as a reference cell containing None, representing that no bits have yet been sampled.

Given a value ℓ sampled using U, we can access approximations of the sampled value using GetBits, which returns the first N bits of the sample as a big-endian integer. To calculate this approximation, GetBits traverses the linked list using a helper function ForceNext, which takes a pointer ℓ to the next cell in the list as an argument. The ForceNext function dereferences the pointer to check whether the next cell in the list exists, and if so, returns it. Otherwise, if the next pointer contains None, then we have reached the end of the list, so ForceNext samples a new Bernoulli value with the command rand 1, which returns 0 or 1 with equal probability. It then appends the sampled value to the end of the linked list. As we will see later (§2), using this GetBits function it is possible to convert this lazily sampled uniform deviate into a form that is compatible with existing libraries for computable real arithmetic, such as CReal [22].

Why is this algorithm correct, and in what sense do the programs U and GetBits sample a uniform value from $[0, 1]$? At a high level, it is a standard fact from measure theory that an infinite sequence of Bernoulli samples, when interpreted as the binary digits of a real number, is equivalent to a uniform sample over $[0, 1]$. But formally applying this argument to justify the code above is challenging. First, the algorithm involves a combination of language features whose semantics are challenging to model and hard to reason about. It directly combines mutable state, dynamically allocated pointers, and random choice. If we wish to further reason about how these samples are used with constructive real libraries like CReal, then we must also consider a setting with higher-order functions. Second, since the bits of the number are sampled lazily, at no point does our program actually obtain an infinite sequence of Bernoulli samples. Yet we would still like to reason about U *as if* it samples a uniform real value $r \in [0, 1]$, in the sense that all calls to GetBits are just returning the digits of the binary expansion of r.

To the best of our knowledge, no techniques from prior work are able to verify an algorithm like the above. While there has been extensive work on reasoning about probabilistic programs and verifying the correctness of sampling algorithms, prior work either restricts to discrete distributions [42, 17, 3] or assumes the existence of primitives for sampling from continuous distributions [40, 14, 32, 31, 50] and works with non-computable real arithmetic.

To address this gap, we present Continuous-Eris, a higher-order separation logic which is capable of verifying the algorithm above, as well as other challenging exact samplers for continuous distributions. This is possible due to a synchronicity between two concepts: continuous distributions are approximable by discrete distributions, and nonterminating programs are approximable by terminating traces. In particular, by adding a variant of *time receipts* [44] to the Eris [2] program logic, we can exploit this coincidence to obtain a program logic with continuous reasoning principles for computable, discrete sampling algorithms.

In summary, we present

- A program logic for verifying computable continuous sampling algorithms using *error credits* and *time receipts*,
- A verified implementation of exact Gaussian, exponential, and Laplace sampling, and
- A formal proof that our Laplace sampler satisfies an accuracy bound used in the differential privacy literature.

```
C′ ℓ1 α1 ℓ2 α2 ≜ let (b1, ℓ1′) := ForceNext ℓ1 α1 in
                 let (b2, ℓ2′) := ForceNext ℓ2 α2 in
                 if b1 < b2 then −1 else
                 if b1 > b2 then 1 else
                 C′ ℓ1′ α1 ℓ2′ α2
CmpU (ℓ1, α1) (ℓ2, α2) ≜ if ℓ1 = ℓ2 then 0 else C′ ℓ1 α1 ℓ2 α2
MaxU2 _ ≜ let x := U _ in
          let y := U _ in
          if CmpU x y < 0 then y else x
```

Figure 2 The maximum of two uniform $[0, 1]$ samples.

The results in this paper have been mechanized [15] in the Rocq proof assistant, building on the Iris separation logic framework [38] and the Coquelicot real analysis library [11].[1]

2 Key Ideas

To illustrate the key concepts of Continuous-Eris, we will verify the MaxU2 program in Figure 2. The program MaxU2 draws two uniform deviates x and y using U and returns their maximum value. To find the maximum, MaxU2 calls the helper function CmpU, which determines the greater of two samples by comparing their bits in order, sampling to a higher precision if needed. A standard exercise in probability theory is to prove that the value returned from the procedure is distributed over $[0, 1]$ with probability density function $\mu_{\max}(x) \triangleq 2x$. In this section we will demonstrate the key principles of Continuous-Eris by formally verifying this fact.

2.1 Primer: Eris

Continuous-Eris is an extension of Eris [2], a program logic for verifying the *approximate correctness* of discrete probabilistic programs. Eris is based on the Iris [38] separation logic framework, and uses techniques from Iris to support reasoning about programs which combine randomness, unbounded recursion, higher-order functions, and state. The algorithms we verify in this paper use all of these features extensively, making Eris a good starting point.

The core construct in Eris is the *error credit*, a separation logic resource $↯(r)$ (for $r \in \mathbb{R}^{\geq 0}$) which can be "spent" to exclude an event whose probability is at most r. The Eris adequacy theorem says that, for any predicate P over values, a proof of $\{↯(\varepsilon)\}\ \mathsf{e}\ \{v.\ P(v)\}$ implies that the probability of the program e terminating with a value that does not satisfy P is at most ε. Importantly, Eris is a partial correctness logic, so this statement implies nothing about the termination probability of e.

Error credit assertions are used with the following rules:

1. *Spending*: $↯(1) \vdash \bot$. The intuition is that a credit is an upper bound on the probability that a specification fails to hold, and an upper bound of 1 on a probability is trivial.

[1] Our development admits as axioms two standard facts about the Riemann integral that are missing from Coquelicot, but is otherwise fully verified. These axioms are Fubini's theorem for continuous functions and that the postcomposition of an integrable function by a continuous function is integrable.

2. *Splitting*: $↯(\varepsilon_1 + \varepsilon_2) \dashv\vdash ↯(\varepsilon_1) * ↯(\varepsilon_2)$. This allows for splitting and joining error credits like other separation logic resources.
3. *Conditioning*: The value of an error credit can be conditioned on the outcome of a random sample drawn with the command $\mathsf{rand}\, N$, provided that the expected value of the error credit is preserved:

$$\{↯(\mathbb{E}_{\mathfrak{U}N}[E])\}\ \mathsf{rand}\, N\ \{t.\ ↯(E(t)) * t \in \{0, \ldots, N\}\} \tag{1}$$

The $\mathsf{rand}\, N$ command returns an integer from the set $\{0, \ldots, N\}$ uniformly, and $\mathfrak{U}N$ is the uniform distribution over that set.

Error credits also inherit the rules of the underlying Iris separation logic, such as the frame rule. Since Iris is an affine logic, excess error credits can be "dropped" or go unused.

To demonstrate how error credits work in practice, we will show how to spend $↯(1/4)$ from a $↯(1/2)$ error budget to exclude an event with probability $1/4$, as captured by the following specification:

$$\{↯(1/2)\}\ \mathsf{rand}\, 3\ \{t.\ ↯(1/4) * t \neq 0\}. \tag{2}$$

First, we use the *splitting* rule to break our credit into two parts.

$$↯(1/2) \vdash ↯(1/4) * ↯(1/4)\,.$$

The first credit will be framed across the call to $\mathsf{rand}\, 3$. For the second credit, set $E(x) \triangleq [x = 0]$, where the *Iverson bracket* $[P]$ has value 1 if P holds, and 0 otherwise. Note that $\mathbb{E}_{\mathfrak{U}3}[E] = 1/4$. So, using the conditioning and frame rules we get

$$\{↯(1/4) * ↯(\mathbb{E}_{\mathfrak{U}3}[E])\}\ \mathsf{rand}\, 3\ \{t. ↯(1/4) * ↯(E(t)) * t \in \{0, \ldots, 3\}\}$$

Now we perform case analysis on t. In the cases where $t \neq 0$, we can continue the proof using the remaining $↯(1/4)$ budget as desired. When $t = 0$ we have that $E(0) = 1$, so that $↯(E(0)) \vdash \bot$ by the spending rule, allowing us to conclude.

2.2 Correctness of Discrete Samplers via Error Credits

As described above, Eris's adequacy theorem allows us to use error credits to upper bound the probability that a postcondition on the return value will fail to hold. At first, it may appear that only being able to prove such upper bounds is a strong restriction on the kinds of probabilistic behaviors we can specify using Eris. However, by considering more general classes of postconditions, one can completely characterize the distribution of values that a program may return. In particular, Marionneau et al. [42] have shown that Eris satisfies a stronger soundness theorem that can be used to prove the correctness of sampling algorithms for discrete distributions.

The key insight is that the *conditioning rule* for the rand command captures the essence of the fact that rand samples from the uniform distribution. Turning this around, Marionneau et al. show that to prove that an arbitrary program e samples from a discrete distribution μ, it suffices to prove a specification about e which looks like a generalized version of (1):

$$\forall F \in \mathcal{B}(T), \{↯(\mathbb{E}_\mu[F])\}\ \mathsf{e}\ \{t.\ ↯(F(t))\} \tag{3}$$

where $\mathcal{B}(T)$ is the set of bounded functions $T \to \mathbb{R}^{\geq 0}$. Informally, this specification states that e behaves like a random event with probability distribution μ, in the sense that we can condition error credits on the outcome of e provided the average value stays the same. To

justify why a specification of the form in (3) suffices to prove that e generates samples with distribution μ, we can consider two instantiations of the above specification for each value $t \in T$: first where F is instantiated by $\overline{I}_t(x) \triangleq [x \neq t]$ and second by $I_t(x) \triangleq [x = t]$. By applying the Eris adequacy theorem to the first instantiation, we get that the probability that $\mu(t)$ samples a value other than t is at most $1 - \mu(t)$, and from the second instantiation we have that the probability that it samples t is at most $\mu(t)$. If e terminates with probability 1, then by combining these inequalities, we have the probability that e returns t is exactly $\mu(t)$, as desired.

When deriving specifications of the form in (3), one is no longer reasoning about the probability of a particular error event. Instead, the proof involves characterizing the expected value of an arbitrary bounded function F applied to the return values of e. Error credits just become a way to systematically track how the steps of e transform that expected value.

A key benefit of this technique is that, not only does it show that a random sampler is correct, it does so in a way that can be used compositionally as part of a larger Eris proof. In particular, once one has a specification of the form (3), one can reason about e in the logic *as if* it sampled from μ. Marionneau et al. [42] prove such specifications for a range of discrete sampling routines and show how they can be used for modular reasoning.

However, because their results and approach were restricted to discrete distributions, they cannot be used directly to verify samplers for continuous distributions like U. Our results show how to generalize their approach to handle continuous distributions.

2.3 From Discrete to Continuous

To state the correctness of continuous samplers, we further generalize the pattern from (3). In place of expectations over discrete distributions, which are countable series, we instead use expectations over continuous distributions, as represented by a Riemann integral over a probability density function.[2] For example, in the case of the uniform distribution over $[0, 1]$, the density function is just $p(x) = 1$, and so the specification pattern that we will prove for U has the form:

$$\forall F \in \mathcal{PC}([0,1]),\ \left\{ \lightning \left(\int_0^1 F(x)\, dx \right) \right\} \mathsf{U}\ ()\ \{v.\ \exists r.\ \lightning (F(r))\ *\ \mathsf{IsReal}\, v\, r\}. \tag{4}$$

Here, $\mathcal{PC}([0,1])$ is the set of bounded, piecewise-continuous functions $[0,1] \to \mathbb{R}^{\geq 0}$, and IsReal is a predicate stating that v is a representation of the real number r (we will define IsReal later in this section). Piecewise continuity ensures that the Riemann integral exists. Like in the discrete setting, equation (4) allows us to condition credits around a call to U *as if* it returned a real number $r \in [0,1]$ uniformly at random. Clients of U will be able to use this conditioning modularly as part of larger correctness and approximate correctness arguments (§4 and §5). Finally, as we will see later in §3, a specification of this form indeed captures what it means for the program to sample uniformly over the continuous distribution, in the sense that all approximants it returns are distributed with the correct probability.

But how can we prove such a specification in the first place? There are several challenges. First, as we previously discussed, at the time U returns, no values have actually been sampled yet, so how can we condition on some value r being selected? Second, the primitive conditioning rule for rand is a discrete sum, yet the above involves an integral. Finally, we need to formally define what it means for the function v to represent the uniform deviate r in Eris, with the IsReal predicate.

[2] We use Riemann integrals instead of Lebesgue integrals because they suffice for the examples we consider here and are simpler to mechanize.

To address these issues, Continuous-Eris combines three ideas from previous logics. First, *pre-sampling tapes* from Clutch [27], allow us to, as a proof device, pre-determine what the sampled values from future rand commands will be. With this we can pre-sample the bits that constitute the real number at the time that U is called. However, Clutch's pre-sampling tapes are not enough on their own, as only allow for pre-sampling a *finite* number of values in advance. The binary expansion of the sampled value r could have an infinite number of digits that are generated by calls to ForceNext.

To overcome this, we exploit the fact that Eris is a partial correctness logic, and adapt the idea of *time receipts* to internalize this partiality in the logic [44]. Because of partiality, in any given execution that we reason about, we only need to consider up to some number k of steps, where k is arbitrary but bounded. Since producing a new digit in the binary expansion takes at least one step, an execution of length k cannot observe more than k bits, so we only need to pre-sample k values to know all of the bits that could eventually be seen in that execution.

Finally, by pre-sampling enough bits, we can get a Riemann sum with the conditioning rule that is arbitrarily close to the integral in (4). Using the thin-air error credit rule proposed by Li et al. [41], we can logically pay for the discretization error that arises in passing from the sum to the integral.

The remainder of this subsection outlines these three ingredients and how they are used to derive the specification (4).

Pre-sampling tapes

Pre-sampling tapes (or *tapes*) were introduced in Clutch [27] as a proof device for representing knowledge about the random events in a program's future. The operational semantics for tapes will be outlined in §3.1, here we focus on the proof rules they enable. In the logic, a tape is a separation logic resource $\alpha \hookrightarrow (N, \vec{n})$ representing the knowledge that all rand commands marked with tape label α (denoted $\mathsf{rand}\ \alpha\ N$) will draw their next $|\vec{n}|$ samples from the finite list $\vec{n}$, in order. This is captured by the rule for rand, which says that the returned value will be the head of the tape.

$$\{\alpha \hookrightarrow (N, n \cdot \vec{n})\}\ \mathsf{rand}\ \alpha\ N\ \{v.\, v = n * \alpha \hookrightarrow (N, \vec{n})\}$$

We can presample a new value to the end of the tape, using the following rule, which allows us to condition error credits similarly to (1):

$$\frac{\{\exists n.\, \alpha \hookrightarrow (N, \vec{n} \cdot n) * \lightning(E(n)) * P\}\ \mathsf{e}\ \{Q\}}{\{\alpha \hookrightarrow (N, \vec{n}) * \lightning(\mathbb{E}_{\mathfrak{U}N}[E]) * P\}\ \mathsf{e}\ \{Q\}} \tag{5}$$

By repeatedly applying this rule, we can pre-sample a finite number of values to a tape, that will then later be consumed by calls to rand.

The gray code in Figure 1 generates a fresh tape α for each uniform deviate returned by U. Intuitively we would like to presample an infinite sequence of bits representing some number r onto this tape: doing so would mean that all future calls to GetBits will deterministically return approximations of a single real number r, and as a consequence, we could condition our error credits based on the value r takes. Note however that presampling tapes can only contain a *finite* list of values. We will now turn our attention to simulating infinite presampling operations using only finite tapes.

Time Receipts

Time receipts were introduced to separation logic by Mével et al. [44] to eliminate reasoning about program behaviors that would take an infeasibly long amount of time to occur (*e.g.* overflowing a 64 bit counter). In Continuous-Eris we adapt this idea to *internalize* the fact that the logic is partial, so that we do not need to reason about diverging executions. Specifically, we introduce an assertion of the form $\mathsf{StepsLeft}(k)$ establishing that k is an upper bound on the steps left that we will reason about in the program, which must always be nonnegative, i.e., $k < 0 \to \mathsf{StepsLeft}(k) \vdash \bot$. This is introduced using the following rule:

$$\frac{\forall k.\, \{\mathsf{StepsLeft}(k) * P\}\, \mathsf{e}\, \{Q\}}{\{P\}\, \mathsf{e}\, \{Q\}} \tag{6}$$

Read from bottom to top, applying this rule gives us $\mathsf{StepsLeft}(k)$ for some arbitrary k. Since we must prove the result for *all* possible values of k, this ensures we consider all finite prefixes of possible executions.

Each step taken by the program generates a new *time receipt* resource $⧗(1)$. Time receipt resources combine additively like error credits, and can be spent to decrease the global runtime bound:

$$\mathsf{StepsLeft}(n) * ⧗(m) \twoheadrightarrow \mathsf{StepsLeft}(n - m).$$

With this in hand, we can construct a resource which provides the *illusion* of an infinite presampling tape. If we know we are reasoning about an execution that can be at most k more steps, and the tape already has k values on it, then any additional values can never be observed. We define an assertion to represent a tape containing an infinite binary sequence, as represented by a function $f : \mathbb{N} \to \{0, 1\}$.

$$\mathsf{InfiniteTape}\ \alpha\ f \triangleq \exists k, \vec{n}.\, \alpha \hookrightarrow (1; \vec{n}) * \mathsf{StepsLeft}(k) * k < |\vec{n}| * \forall i.\, i < k \to \vec{n}[i] = f(i)$$

This assertion says that, under the hood, there is a real tape with at least k samples which match the first k values of f, and we have $\mathsf{StepsLeft}(k)$. For this infinite tape assertion, we have a derived rand rule

$$\{\mathsf{InfiniteTape}\ \alpha\ f\}\ \mathsf{rand}\ \alpha\ 1\ \{v.\, v = f(0) * \mathsf{InfiniteTape}\ \alpha\ (\lambda\, x.\, f(x + 1))\}$$

which says that executing rand α 1 returns the first element $f(0)$ of the sequence, and returns a sequence that has been shifted by 1. The proof of this rule reasons by cases on whether the underlying tape assertion inside of InfiniteTape has any samples left. If it does, it pulls the first sample off of the tape, and also generates a time receipt letting us decrease the StepsLeft bound by 1. When the tape is empty, the StepsLeft term will become negative, allowing us to conclude the proof by the time receipt principle instead. Thus, externally, the specification makes it appear as if α contains an infinite number of bits matching the infinite bitstring f.

Thin-Air Credits

We complete our illusion by deriving a presampling rule that is capable of populating an InfiniteTape with the binary expansion of a real number.

$$\frac{\forall r.\, \{↯(E(r)) * \mathsf{InfiniteTape}\ \alpha\ (\mathrm{bin}\ r) * P\}\, \mathsf{e}\, \{Q\}}{\left\{↯\left(\int_0^1 E(x)\, dx\right) * \alpha \hookrightarrow (1; []) * P\right\}\, \mathsf{e}\, \{Q\}} \tag{7}$$

The function bin : $[0,1] \to (\mathbb{N} \to \{0,1\})$ gives a binary expansion of a real number, and it is not important which of the possible binary expansions this function picks. Deriving this rule requires extending Continuous-Eris with one last ingredient: a rule for generating arbitrarily small error credits "out of thin air", as proposed by Li et al. [41]:

$$\frac{\{\exists \varepsilon > 0.\ \lightning(\varepsilon) \ast P\}\ \mathsf{e}\ \{Q\}}{\{P\}\ \mathsf{e}\ \{Q\}} \tag{8}$$

This rule says that at any time, we can get access to some additional error credit of some arbitrary value. As Li et al. [41] show, this rule can be added without affecting the soundness of Eris through a limiting argument taking $\varepsilon \to 0$.

To derive the specification (7), we first apply the rule (6) to get $\mathsf{StepsLeft}(k_1)$ for some k_1. Next, we apply (8) to get some credit $\varepsilon > 0$, in addition to the $\int_0^1 E(x)\,dx$ error credit we start with. From Riemann integrability of E, there exists $\delta > 0$, such that any Riemann sum over E with partition width smaller than δ will be within ε of $\int_0^1 E(x)\,dx$. Let k_2 be such that $1/2^{k_2} < \delta$. Set $k = \max(k_1, k_2)$. We now apply the basic presampling rule (5) k times to get k bits on the tape α. If we interpret these bits as the first k bits of the fractional representation of a real number, then unfolding the expected value sums from the repeated use of (5), we get a sum for the expected value of the resulting error credit that can be rewritten as

$$\sum_{i=0}^{2^k-1} E\left(\frac{i}{2^k}\right) \cdot \frac{1}{2^k}$$

This is a Riemann sum over $[0,1]$ with partition width $1/2^k$, thus it is less than the $\varepsilon + \lightning\left(\int_0^1 E(x)\,dx\right)$ credit that we have. Since Eris is an affine logic that allows us to "throw away" excess error credits, we are therefore done.

With this infinite tape pre-sampling rule, we are finally able to derive (4), the specification for U. We define the predicate $\mathsf{IsReal}\ v\ r$ to state that the heap location v points-to a linked list of bits and an $\mathsf{InfiniteTape}$, such that the bits on the list together with the bits on the infinite tape form a binary sequence for r. Executing U gives us access to a heap location and empty tape, and we can apply (7) to establish the IsReal predicate and the error credits in the postcondition of (4).

Summary

In this subsection, we have seen how a few logical ingredients allow us to derive a continuous analogue of the specification style previously proposed by Marionneau et al. [42] for discrete samples. In particular, extending Eris with time receipts allowed us to extend Clutch's finite pre-sampling tapes into a mechanism that behaves like an infinite tape. Next we will explore how to use this specification in client code, using careful credit conditioning to verify algorithms based on uniform real samples.

2.4 Verified Maximum Sampler

Now we can now return to the verification of $\mathsf{MaxU2}$ from Figure 2. We will do this in explicit detail as it illustrates the core techniques we will be making use of in our more complex examples. Recall that the density of the maximum of two uniforms is given by $\mu_{\mathsf{MaxU2}}(x) = 2x$. We can state a continuous correctness theorem for $\mathsf{MaxU2}$ as:

$$\forall G \in \mathcal{PC}([0,1]),\ \left\{\lightning\left(\int_0^1 G(x) \cdot 2x\,dx\right)\right\}\ \mathsf{MaxU2}\ ()\ \{v.\ \exists r.\ \mathsf{IsReal}\ v\ r \ast \lightning(G(r))\}. \tag{9}$$

In other words, given some G, and $\varepsilon_0 \triangleq \int_0^1 G(x) \cdot 2x\,dx$ error credits, we want to execute the body of MaxU2 and ensure that the return value of MaxU2 approximates a real number r, such that we end up with $↯(G(r))$ credits leftover. To prove this, we will apply the specification for U (4) at each of the uniform sampling statements, instantiating the function F from that specification with appropriate credit conditioning functions.

For the first call to U, we instantiate F with the function ε_1 defined by $\varepsilon_1(x) \triangleq \int_0^1 G(\max(x, y))\,dy$. Let r_1 be the real number we obtain in the post-condition of the rule. Then, for the second call to U we instantiate F with ε_2 defined by $\varepsilon_2(y) \triangleq G(\max(r_1, y))$. Call the result of this call r_2.

At this point there are two parts left to the proof. The first is to show that the iterated integral we obtain from applying this rule twice with these choices of F is equivalent to ε_0, the error credits we have in the precondition of (9). We have

$$\begin{aligned}
&\int_0^1 \int_0^1 G(\max(x, y))\,dx\,dy \\
&= \int_0^1 \int_0^1 [x < y] G(y)\,dx\,dy + \int_0^1 \int_0^1 [y < x] G(x)\,dx\,dy \\
&= 2 \int_0^1 G(z) \int_0^1 [w < z]\,dw\,dz \\
&= \int_0^1 G(z) \cdot 2z\,dz \\
&= \varepsilon_0
\end{aligned}$$

The second part of the proof is to show that the code in fact computes the maximum $\max(r_1, r_2)$. But this part of the reasoning involves no probability theory, as it amounts to reasoning about the bit-by-bit comparison done by traversing the linked-list of bits in the two numbers inside CmpU using ForceNext. Readers unfamiliar with Iris may be concerned that the recursive calls in C′ are not structurally decreasing in any argument. Formally speaking, we verify C′ using the Löb induction rule from Eris.

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

The Löb induction rule allows us to assume that the specification for C′ holds at every recursive call in its body. In a call to C′, if the leading bits are different, then the real number starting with 1 is greater than or equal to the one starting with 0. If the leading digits match, the program C′ will recurse and we conclude by the Löb induction hypothesis.

The proofs for subsequent more sophisticated sampling algorithms follow a similar pattern: first we choose the right instantiations of the function F for each random sample the algorithm draws, and then we prove that the resulting iterated integrals are equivalent to the integral over the density of the distribution the algorithm is generating samples from. Then, after the values are sampled, we are left with traditional reasoning about linked lists or other bit manipulations of the sampled values, for which separation logic is well-suited. With Continuous-Eris, proofs of this format (including the more complex sampler in §4 and §5) can be implemented formally inside the Rocq proof assistant.

$$
\begin{aligned}
\mathsf{OfZ}\ z\ p &\triangleq z \ll p \\
\mathsf{Radd}\ r_1\ r_2\ p &\triangleq \mathsf{let}\ z := r_1\ (p+2) + r_2\ (p+2)\ \mathsf{in} \\
&\quad\ (z/4) + (z\,\%\,4)/2 \\
\mathsf{Rneg}\ r\ p &\triangleq -(r\ p) \\
\mathsf{Rscal}\ r\ z\ p &\triangleq r\ (p - z) \\
\mathsf{RofU}\ u\ p &\triangleq \mathsf{if}\ p \leq 0\ \mathsf{then}\ 0\ \mathsf{else}\ \mathsf{GetBits}\ u\ p\ 0 \\
\mathsf{Rcmp}\ r_1\ r_2\ p &\triangleq \mathsf{let}\ n_1 := r_1\ p\ \mathsf{in} \\
&\quad\ \mathsf{let}\ n_2 := r_2\ p\ \mathsf{in} \\
&\quad\ \mathsf{if}\ n_1 + 2 < n_2\ \mathsf{then}\ -1\ \mathsf{else} \\
&\quad\ \mathsf{if}\ n_2 + 2 < n_1\ \mathsf{then}\ 1\ \mathsf{else} \\
&\quad\ \mathsf{Rcmp}\ r_1\ r_2\ (p+1)
\end{aligned}
$$

Figure 3 The Continuous-Eris constructive real library.

2.5 Interlude: Constructive Real Arithmetic

While our lazy bit list representation of $[0, 1]$ suffices for drawing and comparing uniform samples, clients of U such as the Gaussian or Laplace sampler will require us to represent samples over the entire real line. Before we go deeper into these more complex sampling procedures, we will discuss a simple implementation of *constructive real arithmetic*, which is compatible with the lazy bit lists of U, and which we have verified in Continuous-Eris.

There are a number of options in the literature for representing lazily defined real numbers, such as signed bit sequences [56], continued fractions [54], or sequences of dyadic approximants [43]. Our implementation is an adaptation of the CReal library [22], which is based on the latter technique.

We represent a real number r using functions $s_r : \mathbb{Z} \to \mathbb{Z}$, which return integer approximations of r to a specified accuracy. Formally, we define a predicate $\mathrm{ApproxTo}(A, z, r)$ which says that $A \in \mathbb{Z}$ approximates r to within a degree of precision specified by $z \in \mathbb{Z}$:

$$\mathrm{ApproxTo}(A, z, r) \triangleq |A - r \cdot 2^z| \leq 1. \tag{10}$$

We say then that a function s_r is a valid representation of r if $\mathrm{ApproxTo}(s_r(z), z, r)$ holds for all z. This condition ensures that as $z \to \infty$, the sequence $s_r(z)/2^z$ converges to r.

Figure 3 depicts our implementation of a subset of operations from CReal. With the exception of Rcmp, each operator returns a function from desired precision $p \in \mathbb{Z}$ to an integer approximant. Our library includes the following:

- OfZ z: This converts the constant integer z to a real whose sequence is obtained by bit-shifting z.
- Radd r_1 r_2: Addition between r_1 and r_2. The first line adds the $(p+2)^{\mathrm{nd}}$ approximants for r_1 and r_2 together, and the second line divides this sum by four (rounding to the nearest integer).
- Rneg r_1: Negates a real number by negating each approximant.
- Rscal r z: Uses a bit shift to approximate the real number $r/2^z$ for $z \in \mathbb{Z}$. CReal defines a more complicated procedure for general multiplication between real numbers, but scaling by powers of two will suffice for our purposes.
- RofU u: Convert a partly sampled uniform deviate into a lazy real number. The GetBits function was defined in Figure 1.

$$\mathsf{RofBZU}\ b\ z\ u \triangleq \mathsf{let}\ \mathsf{abs} := \mathsf{Radd}\ (\mathsf{OfZ}\ z)\ (\mathsf{RofU}\ u)\ \mathsf{in}\\ \mathsf{if}\ b = 0\ \mathsf{then}\ \mathsf{abs}\ \mathsf{else}\ \mathsf{Rneg}\ \mathsf{abs}$$

Figure 4 Conversion from boolean-integer-uniform triples into constructive reals.

Like CmpU, the function Rcmp compares two real numbers by iteratively comparing approximants until they are sufficiently far apart to determine their inequality.[3] In our Rocq development, we specify the correctness of these operators using the following Continuous-Eris predicate:

$$\mathrm{IsApprox}\ (v : \mathit{Val})\ (x : \mathbb{R}) \triangleq \exists I.\ I * \Box \forall (p : \mathbb{Z}).\ \{I\}\ v\ p\ \{A.\ I * \mathrm{ApproxTo}(A, p, r)\}$$

In other words, IsApprox states that a Continuous-Eris function v can be executed, using some set of resources I, in order to return an approximation of r with any precision p. Using this representation, it is straightforward to verify the correctness of the arithmetic operators in our library, such as the addition function.

$$\{\mathrm{IsApprox}\ v_x\ x * \mathrm{IsApprox}\ v_y\ y\}\ \mathsf{Radd}\ v_x\ v_y\ \{v.\ \mathrm{IsApprox}\ v\ (x + y)\}$$

The correctness here depends principally on using the triangle inequality to relate the $(p+2)^{\mathrm{nd}}$ approximants of x and y to the p^{th} approximant of $x + y$.

We can also lift the lazily uniform samples of U into constructive reals.

$$\{\mathsf{IsReal}\ u\ x\}\ \mathsf{RofU}\ u\ \{v.\ \mathrm{IsApprox}\ v\ x\}$$

In this proof, the I in IsApprox is instantiated with IsReal u x, which allows us to compute arbitrarily precise approximants for x using GetBits.

Finally, we will derive one last program for constructing values in $\mathbb{R}$ using these primitive operators. The equation $\mathrm{ToReal}(b, z, r) \triangleq (-1)^b(z + r)$ will serve as our canonical mapping from boolean-integer-uniform triples (b, z, r) to real numbers. The program RofBZU in Figure 4 takes as input a boolean-integer-real triple, and returns the corresponding constructive real. We will use this program in the verification of the Gaussian and Laplace samplers, and also our soundness theorem.

We have now seen how to establish and use a continuous analogue of the specification style that Marionneau et al. [42] previously demonstrated for verifying discrete samplers. But what does this specification actually mean? In the next section, we prove an adequacy theorem that connects this specification to the semantics of the program.

3 Soundness

This section describes our soundness results for Continuous-Eris. Before we can state the soundness results, we first need to define the semantics of the language we are considering formally. Then, we prove that the basic adequacy theorem for Eris Hoare triples also holds for Continuous-Eris. Finally, we prove an analogue of Marionneau et al.'s [42] adequacy theorem for the correctness of continuous samplers.

[3] Note that the comparison function does not terminate when the arguments are equal. Unlike the comparison between two uniform deviates, it is possible to write programs where comparisons loop with probability greater than zero.

3.1 RandML Language

$$
\begin{aligned}
v \in \mathit{Val} &\triangleq z \in \mathbb{Z} \mid b \in \mathbb{B} \mid () \mid \ell \in \mathit{Loc} \mid \kappa \in \mathit{Label} \mid \mathsf{rec}\,\mathsf{f}\,\mathsf{x} = e \mid (v, w) \mid \mathsf{inl}\,v \mid \mathsf{inr}\,v \\
e \in \mathit{Expr} &\triangleq v \mid \mathsf{x} \mid e_1\ e_2 \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1 \ll e_2 \mid e_1 \gg e_2 \mid \ldots \mid \\
&\quad \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid (e_1, e_2) \mid \mathsf{fst}\,e \mid \mathsf{snd}\,e \mid \mathsf{inl}(e) \mid \mathsf{inr}(e) \mid \\
&\quad \mathsf{match}\ e\ \mathsf{with}\ \mathsf{inl}\,v \Rightarrow e_1 \mid \mathsf{inr}\,w \Rightarrow e_2\ \mathsf{end} \mid \mathsf{allocn}\,e_1\ e_2 \mid\ !\,e \mid e_1 \leftarrow e_2 \mid \\
&\quad \mathsf{rand}\,e \mid \mathsf{rand}\,e_1\,e_2 \mid \mathsf{tape}\,e \\
K \in \mathit{Ectx} &\triangleq\ - \mid e\,K \mid K\,v \mid \mathsf{allocn}\,K \mid\ !\,K \mid e \leftarrow K \mid K \leftarrow v \mid \mathsf{rand}\,K \mid \ldots \\
t \in \mathit{Tape} &\triangleq \{(N, \vec{n}) \mid N \in \mathbb{N} \wedge \vec{n} \in \mathit{List}\mathbb{N}_{\leq N}\} \\
\sigma \in \mathit{State} &\triangleq (\mathit{Loc} \xrightarrow{\text{fin}} \mathit{Val}) \times (\mathit{Label} \xrightarrow{\text{fin}} \mathit{Tape}) \qquad \rho \in \mathit{Cfg} \triangleq \mathit{Expr} \times \mathit{State}
\end{aligned}
$$

Figure 5 Syntax of RandML.

RandML is the same language as used in Eris. The syntax of RandML is depicted in Figure 5. The language includes primitives for higher-order programming such as higher-order references and function closures, as well as generic recursion using recursive let bindings. RandML's sole source of randomness is the $\mathsf{rand}\,N$ command seen in the previous section. Notably, RandML does *not* include primitives for sampling from continuous distributions. Instead, we implement and verify computable versions of these algorithms.

Because the language only has discrete sampling as a primitive, we are able to give an operational semantics that does not require measure theory. To account for the possibility of nontermination, we use the monad of discrete *subdistributions*.

▶ **Definition 1** (Subdistribution). *A discrete subdistribution over a countable set A is a function $\mu : A \to [0, 1]$ such that $\sum_{a \in A} \mu(a) \leq 1$, sometimes also called a* probability mass function *or PMF. We let $\mathcal{D}(A)$ denote the set of all subdistributions over A. The discrete Giry monad [26] is a monad on $\mathcal{D}$ where the return δ_x is the Dirac distribution at x and*

$$(\mu >\!\!>\!\!= f)(b) \triangleq \sum_{a \in A} \mu(a) \cdot f(a)(b) \tag{11}$$

Given a subset $\phi \subset A$ and $\mu \in \mathcal{D}(A)$, the probability of ϕ under μ, denoted $\Pr_\mu[\phi]$, is given by

$$\Pr_\mu[\phi] = \sum_{a \in \phi} [\mu(a)]$$

We give a small step operational semantics to RandML in terms of subdistributions, following Eris. Namely, we define the semantics for a single step of reduction by a monadic function on subdistributions $\mathsf{step} : \mathit{Cfg} \to \mathcal{D}(\mathit{Cfg})$, where a configuration Cfg is a pair of an expression Expr and a state State. For deterministic cases (such as arithmetic operations or writing to the store) we describe their posterior subdistribution using the monadic return. Error cases such as ill-typed applications return the zero subdistribution $\mathbf{0}$. The $\mathsf{rand}\ N$ primitive draws samples from $\mathfrak{U}N$, the uniform distribution over $\{0, \ldots, N\}$, and leaves the state σ unchanged.

$$\mathsf{step}(\mathsf{rand}\ N, \sigma) \triangleq \begin{cases} \mathfrak{U}N >\!\!>\!\!= \lambda n.\, \delta_{(n,\sigma)} & \text{if } N \in \mathbb{N} \\ \mathbf{0} & \text{otherwise} \end{cases}$$

The semantics also models the presampling tapes we saw in §2. A tape consists of a *label* α, a *bound* $N \in \mathbb{N}$, and a finite sequence of *presamples* $\vec{n}$. When rand includes the optional tape label parameter, it will deterministically pop a sample from the tape if the tape is non-empty, and otherwise draws a fresh random sample.

$$\mathsf{step}(\mathsf{rand}\ \alpha\ N, \sigma) \triangleq \begin{cases} \delta_{(n,\sigma[\alpha \mapsto (N,\vec{n})])} & \text{if } N \in \mathbb{N} \text{ and } \alpha \mapsto (N, n \cdot \vec{n}) \in \sigma \\ \mathfrak{U}N >\!\!>\!= \lambda n.\, \delta_{(n,\sigma)} & \text{if } N \in \mathbb{N} \text{ and } \alpha \mapsto (N, n \cdot \vec{n}) \notin \sigma \\ \mathbf{0} & \text{otherwise} \end{cases}$$

Tape labels are dynamically allocated using the tape primitive in a similar fashion to heap locations. However, the RandML language includes no language primitives for adding samples onto the tape. Rather, samples are added to the end of the tape during a proof by applying the presampling rules we saw previously. The soundness proof for the logic includes an *erasure theorem*, showing that every program has the same semantics as one with all tape operations omitted.

The semantics of a program is the limit of finite executions. Concretely, the subdistribution $\mathsf{exec}_n : \mathit{Cfg} \to \mathcal{D}(\mathit{Val})$ represents the subdistribution of values reached after n steps of execution.

$$\mathsf{exec}_n(e,\sigma) \triangleq \begin{cases} \delta_e & \text{if } e \in \mathit{Val} \\ \mathsf{step}(e,\sigma) >\!\!>\!= \mathsf{exec}_{n-1} & \text{if } e \notin \mathit{Val} \text{ and } n > 0 \\ \mathbf{0} & \text{otherwise} \end{cases} \tag{12}$$

Any configuration that has not reached a value after n steps does not contribute its probability to exec_n. The subdistribution of return values $\mathsf{exec}(\rho)$ is defined at each point to be the limit of exec_n as $n \to \infty$.

3.2 Soundness of the Logic

We proved the following adequacy theorem of Continuous-Eris, which is the same as that of Eris:

▶ **Theorem 2** (Adequacy). *For any pure predicate on values P, if we can prove $\{\lightning(\varepsilon)\}\, e\, \{v.\, P\ v\}$ in Continuous-Eris, then for all states σ, we have* $\Pr[\mathsf{exec}(e,\sigma) \notin P] \leq \varepsilon$.

Recall that Continuous-Eris features two proof rules not found in the original Eris: thin-air credits and time receipts. The thin-air credit rule was proved sound by Li et al. [41] for an extension to Eris for concurrent programs, and we have backported their proof technique to Continuous-Eris. For time receipts, we exploit the fact that the existing proof of soundness for Eris is based on step-indexing: since the logic is partial, the proof works by explicitly considering executions of up to n steps for each value of n. Thus, all we need to do is track this number of steps remaining in the execution we are considering using a ghost resource, in order to model the StepsLeft assertion.

3.3 Adequacy for Continuous Samplers

To formally capture the correctness of continuous samplers, we prove a version of the adequacy theorem tailored to continuous samplers returning lazy reals satisfying the IsApprox predicate. Stating this is subtle: we want to specify what it means for a program to lazily return digits

$$\mathsf{Checker}\ \mathsf{e}\ N\ D \triangleq \mathsf{Rcmp}\ e\ (\mathsf{Rscal}\ (\mathsf{OfZ}\ N)\ D)\ 0$$

Figure 6 Definition of the Checker program.

from a real number, but clearly this will depend on the RandML representation of a real. The proof of adequacy by Marionneau et al. [42] avoids this issue as their samplers all return simple, first-order datatypes such as $\mathbb{Z}$ or booleans, all of which have canonical representatives in RandML.

The key idea of our approach to this problem is to capture the behavior of a continuous sampler by analyzing the *cumulative distribution function* (CDF) it converges to. An absolutely continuous probability (sub-)distribution over the reals is characterized by a *density function* $h \in \mathcal{PC}(\mathbb{R})$, such that $\int_{-\infty}^{\infty} h(x)\,dx \leq 1$ and whose definite integrals $\int_a^b h(x)\,dx$ for $a \leq b$ correspond to the probability of sampling a value in the interval $[a, b]$. Its corresponding CDF H can be obtained by the equation $H(r) = \int_{-\infty}^{r} h(x)\,dx$.

We can observe the CDF of a RandML real random sampler by linking it against the program Checker in Figure 6. The Checker program takes in two arguments: the sampler e to be analyzed, and two integers N and D. Intuitively, Checker samples from e and compares the sampled value to the dyadic rational number $N/2^D$. The Checker program will diverge if the sampled value is exactly equal to $N/2^D$, otherwise it returns -1 or 1 depending on whether e returns a constructive real less than or greater than $N/2^D$, respectively. Our adequacy theorem describes samplers in terms of their observable behavior when linked against this checker program, stated more precisely in Theorem 3.

▶ **Theorem 3** (Partial Adequacy of Constructive Real Samplers). *Let $h \in \mathcal{PC}(\mathbb{R})$ be a density function. If for all integrable $F \in \mathcal{PC}(\mathbb{R})$ we can prove*

$$\left\{ \lightning \left(\int_{-\infty}^{\infty} F(x)h(x)\,dx \right) \right\} \mathsf{e} \left\{ v.\ \exists r.\ \lightning (F\ r)\ *\ \mathit{IsApprox}\ v\ r \right\}, \tag{13}$$

then for all states σ and integers B, C we have

1. $\Pr[\mathsf{exec}(\mathsf{Checker}\ \mathsf{e}\ B\ C, \sigma) \neq 1] \leq \int_{-\infty}^{B/2^C} h(x)\,dx$
2. $\Pr[\mathsf{exec}(\mathsf{Checker}\ \mathsf{e}\ B\ C, \sigma) \neq -1] \leq \int_{B/2^C}^{\infty} h(x)\,dx$

This theorem relates the behavior of the Checker program with the CDF of the target continuous distribution. To understand this theorem, recall that $\int_{-\infty}^{B/2^C} h(x)\,dx$ is the CDF of the distribution at $B/2^C$, *i.e.* it is the probability that a value sampled from the distribution will be less than or equal to $B/2^C$. The first inequality says that the probability that Checker terminates with a value other than 1 is at most this CDF value. The second inequality gives us an analogous result about the probability that the sampled value is greater than $B/2^C$.

These inequalities give us a good sense of the approximate behavior of sampler programs when linked against Checker, however they do not completely characterize the CDF of e because Continuous-Eris is a partial logic. It is possible that the sampling program fails to terminate with probability greater than zero.[4] By adding hypotheses to Theorem 3, we obtain a stronger result.

[4] This is only possible if the sampler program e fails to terminate. The Checker program only fails to terminate when e returns the exact value $B/2^C$, which can be shown to happen with probability at most zero by instantiating Equation (13) with the indicator function on $\{B/2^C\}$.

```
R N x ≜ let y := U in
        if CmpU y x < 0 then R (N + 1) y else N
H _ ≜ let x := U in
      if RleHalf x then R 0 x % 2 = 1 else true
```

Figure 7 The decreasing uniform sequence trial (R) and the Bernoulli trial with parameter $e^{-1/2}$ (H).

▶ **Theorem 4** (CDF Adequacy of Constructive Real Samplers). *Under the assumptions of Theorem 3, if the checker program terminates almost-surely (*i.e. *for all B, C, and σ we have $\sum_{v \in Val} \mathsf{exec}(\mathsf{Checker}\ \mathsf{e}\ B\ C, \sigma)(v) = 1$) and h is a probability density (*i.e. $\int_{-\infty}^{\infty} h(x)\,dx = 1$*) then for all states σ and integers B, C, the probability that* Checker e B C *returns -1 is equal to the CDF of h at the point $B/2^C$:*

$$\Pr[\mathsf{exec}(\mathsf{Checker}\ \mathsf{e}\ B\ C, \sigma) = -1] = \int_{-\infty}^{B/2^C} h(x)\,dx. \tag{14}$$

Since the set of dyadic integers is dense in $\mathbb{R}$, characterizing the CDF at these points suffices to characterize the distribution of e completely (see Billingsley [8], theorem 14.1).

In our Rocq development, we have formally verified both of these adequacy statements. The proof of Theorem 4 is a consequence of Theorem 3, using the additional termination hypotheses in order to establish (14). For the programs we will present in sections §4 and §5, we do not prove their almost-sure termination, though Karney [39] proves on paper that these algorithms terminate almost-surely. Separate program logics such as Total Eris [2] or Tachis [29], which cover the same source language, are capable of proving this fact. For the remainder of this paper, we will focus on establishing the core specification (13), and leave the other side conditions in Theorem 4 to future work.

4 Verified Gaussian Sampler

In this section, we will verify a sampler for the continuous Gaussian distribution using Continuous-Eris. We will use algorithms from [39]. To the best of our knowledge, these algorithms have never been formally verified. Karney's [39] sampler is split into a number of subroutines, and our proof follows the same modular structure by verifying each subroutine.

4.1 Bernoulli Negative Half-Exponential

The first component of Karney's [39] algorithm is a routine H for drawing a sample from the Bernoulli($e^{-1/2}$) distribution. This in turn builds upon a routine R which repeatedly draws a sequence of independent, uniform samples $r_1, r_2, r_3, \ldots$ from $[0, 1]$. The routine stops when it draws an r_{i+1} that is greater than r_i, at that point, it returns i, the length of the decreasing sequence it generated.

Verifying the Decreasing Reals Trial

The principle behind this algorithm is sometimes known as *Von Neumann's Technique* [53]. Von Neumann's insight is that the probability that this decreasing sequence has length n is $x^n/n! - x^{n+1}/(n+1)!$ – two subsequent terms in the Taylor series of e^{-x}.

In general, $\mathsf{R}\ N\ x$ is distributed over $\mathbb{N}$ as a version of $\mathsf{R}\ 0\ x$ that has been shifted rightwards by N:

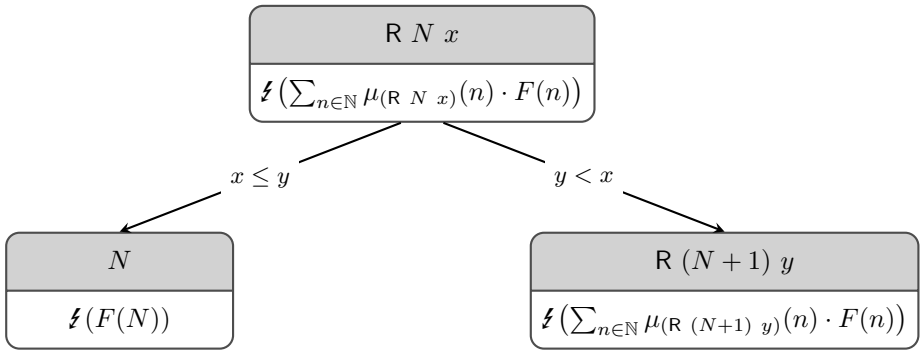
$$\mu_{(\mathsf{R}\ N\ x)}(n) \triangleq [N \le n] \cdot \left(\frac{x^{n-N}}{(n-N)!} - \frac{x^{n-N+1}}{(n-N+1)!} \right) \tag{15}$$

Note that although this sampler draws continuous samples internally, what it *returns* is a sample from a *discrete* distribution over $\mathbb{N}$. Thus, the specification we prove for it uses a series for computing the expected value of credits, instead of an integral. Let $\mathcal{B}(\mathbb{N})$ denote the set of bounded functions $\mathbb{N} \to \mathbb{R}^{\ge 0}$. We seek to prove the following specification for all $F \in \mathcal{B}(\mathbb{N})$, $N \in \mathbb{N}$, $l \in \mathit{Val}$, and $x \in [0,1]$:

$$\left\{ \begin{array}{l} \mathsf{IsReal}\ l\ x * 0 \le x \le 1 * \\ \lightning \left(\sum_{n \in \mathbb{N}} \mu_{(\mathsf{R}\ N\ x)}(n) \cdot F(n) \right) \end{array} \right\} \mathsf{R}\ N\ x \left\{ n, \begin{array}{l} \lightning (F(n)) * \\ \mathsf{IsReal}\ l\ x \end{array} \right\} \tag{16}$$

At a high level we will prove this in a similar way to §2.4, using specification (4) at the call to U to redistribute the error credits based on the value of y.

The challenge in proving (16) lies in correctly conditioning the credits after each probabilistic sampling from U. We proceed by Löb induction, using HT-REC. The execution of $\mathsf{R}\ N\ x$ first evaluates the sampling statement U, branches on its result, and either returns N (requiring $\lightning(F(N))$ error credits for the postcondition) or makes a recursive call (requiring enough error credits to pay for the precondition of (16) via HT-REC). Schematically, we want to condition the credits after sampling from U as follows:



With this in mind, we choose the following function g to condition the error credits:

$$g(F, N, x, y) \triangleq [x \le y] \cdot F(N) + [y \le x] \left(\sum_{m \in \mathbb{N}} \mu_{(\mathsf{R}\ (N+1)\ y)}(m) \cdot F(m) \right) \tag{17}$$

After sampling from U we obtain a lazy real y and $\lightning(g(F,N,x,y))$ error credits. However, note that at this point we have no further information about y, which we need to evaluate the comparison $\mathsf{CmpU}\ y\ x$. Therefore, we perform a case distinction on $(x \le y) \vee (y \le x)$, which can be used to cancel out the corresponding Iverson bracket in (17) and keep executing the program symbolically[5]. In the first case, we will own $\lightning(F(N))$, and, by symbolic execution, the program will evaluate to the else branch and return N, so the postcondition is satisfied. In the second case, we will own $\lightning\left(\sum_{m \in \mathbb{N}} \mu_{(\mathsf{R}\ (N+1)\ y)}(m) \cdot F(m)\right)$, and we will reach the recursive call to $\mathsf{R}\ (N+1)\ y$, at which point we can use the inductive hypothesis since we have the correct amount of credits available. It remains to show that the expected value of g is equal to our initial quantity of starting credits.

[5] The fact that the Iverson brackets are overlapping at the point $y = x$ is a matter of convenience.

$$\begin{aligned}
\mathsf{G}\ e\ N &\triangleq \mathsf{if}\ e\ ()\ \mathsf{then}\ \mathsf{G}\ e\ (N+1)\ \mathsf{else}\ N\\
\mathsf{I}\ e\ N &\triangleq (N=0)\ \|\ (e\ ()\ \&\&\ \mathsf{I}\ e\ (n-1))\\
\mathsf{Z}\ _ &\triangleq \mathsf{let}\ k := \mathsf{G}\ \mathsf{H}\ 0\ \mathsf{in}\\
&\qquad \mathsf{if}\ \mathsf{I}\ (\mathsf{H}\ (k*(k-1)))\ \mathsf{then}\ k\ \mathsf{else}\ \mathsf{Z}\ ()
\end{aligned}$$

Figure 8 The integer part of the Gaussian sampler.

$$\sum_{n\in\mathbb{N}} \mu_{(\mathsf{R}\ N\ x)}(n)\cdot F(n) = \int_0^1 g(F,N,x,y)\,dy \tag{18}$$

The proof of this fact amounts to an application of Fubini's theorem in a similar style to §2.4. Here, the boundedness of F ensures absolute and uniform convergence of the limits in (18), allowing their exchange.

Bernoulli Negative Half-Exponential

Finally, our specification for R can be used to verify the correctness of the half-exponential Bernoulli sampler.[6] We seek to prove that for any $F : \{\mathsf{true}, \mathsf{false}\} \to \mathbb{R}^{\geq 0}$,

$$\left\{ \lightning\left(e^{-1/2}F(\mathsf{true}) + (1-e^{-1/2})F(\mathsf{false})\right)\right\}\ \mathsf{H}\ \{b, \lightning(F(b))\} \tag{19}$$

The uniform sample in H is handled by the credit distribution

$$\left[\frac{1}{2} < x\right]F(\mathsf{true}) + \left[x \leq \frac{1}{2}\right]\left(\sum_{m\in\mathbb{N}} \mu_{(\mathsf{R}\ 0\ x)}(m)\cdot F(m\%2 = 1)\right) \tag{20}$$

In the case that $1/2 < x$ we gain the correct amount of credits to return $F(\mathsf{true})$, and if not, we have enough to apply the specification for R. The function RleHalf is a version of CmpU which compares a lazy real sample against the constant value $1/2$, and is verified in a similar manner. In our development, we split the series in expression (20) into even and odd terms, at which point we can apply Von-Neumann's trick.

$$\sum_{n\in\mathbb{N}} \frac{(1/2)^{2n}}{(2n)!} - \frac{(1/2)^{2n+1}}{(2n+1)!} = \sum_{n\in\mathbb{N}} \frac{(-1/2)^n}{n!} = e^{-1/2}$$

4.2 Integer Gaussian Part

The next component of the algorithm is a routine to sample the *integer* part of the Gaussian sample, as shown in Figure 8. Specifically, the routine Z draws from the distribution over the nonnegative integers with PMF

$$\mu_{\mathsf{Z}}(k) \triangleq e^{-k^2/2}/\mathcal{N}_{\mathsf{Z}} \qquad \mathcal{N}_{\mathsf{Z}} \triangleq \sum_{k=0}^{\infty} e^{-k^2/2} \tag{21}$$

[6] In our implementation of H, we must handle the first iterate of this process separately, since CmpU can only compare two uniform deviates against each other, and so we need to use a specialized program RleHalf to compare a uniform deviate against the constant $1/2$. We choose to do it this way instead of lifting to constructive real library in order to mimic more closely Karney's implementation.

$$
\begin{aligned}
\mathsf{C}\ M &\triangleq \mathsf{let}\ m := \mathsf{rand}\ M\ \mathsf{in} \\
&\quad\ \mathsf{if}\ m = 0\ \mathsf{then}\ -1\ \mathsf{else\ if}\ m = 1\ \mathsf{then}\ 0\ \mathsf{else}\ 1 \\
\mathsf{A}\ k\ x &\triangleq \mathsf{let}\ f := \mathsf{C}\ k\ \mathsf{in} \\
&\quad\ \mathsf{let}\ r := \mathsf{U}\ \mathsf{in} \\
&\quad\ (f = 0)\ \|\ (f = -1\ \&\&\ \mathsf{CmpU}\ x\ r < 0) \\
\mathsf{S}\ k\ x\ y\ N &\triangleq \mathsf{let}\ z := \mathsf{U}\ \mathsf{in} \\
&\quad\ \mathsf{if}\ (\mathsf{CmpU}\ y\ z < 0\ \|\ \mathsf{A}\ k\ x)\ \mathsf{then}\ N\ \mathsf{else}\ (\mathsf{S}\ k\ x\ z\ (N+1)) \\
\mathsf{S}'\ k\ x &\triangleq \mathsf{let}\ z := \mathsf{U}\ \mathsf{in} \\
&\quad\ \mathsf{if}\ (\mathsf{CmpU}\ x\ z < 0\ \|\ \mathsf{A}\ k\ x)\ \mathsf{then}\ 0\ \mathsf{else}\ (\mathsf{S}\ k\ x\ z\ 1) \\
\mathsf{B}\ k\ x &\triangleq\ (\mathsf{S}'\ k\ x)\,\%\,2 = 0 \\
\mathsf{N}'\ \_ &\triangleq \mathsf{let}\ k := \mathsf{Z}\ ()\ \mathsf{in} \\
&\quad\ \mathsf{let}\ x := \mathsf{U}\ \mathsf{in} \\
&\quad\ \mathsf{if}\ \mathsf{I}\ (\mathsf{B}\ k\ x)\ (k+1)\ \mathsf{then}\ (k,x)\ \mathsf{else}\ \mathsf{R}\ \mathsf{N}'() \\
\mathsf{N}\ \_ &\triangleq \mathsf{let}\ (z,u) := \mathsf{N}'\ ()\ \mathsf{in} \\
&\quad\ \mathsf{let}\ b := \mathsf{rand}\ \ 1\ \mathsf{in} \\
&\quad\ \mathsf{RofBZU}\ b\ z\ u
\end{aligned}
$$

Figure 9 The Gaussian sampler (N).

The implementation uses helper functions G (a geometric trial, parameterized by a function e that generates Bernoulli samples) and I (which draws N Bernoulli samples using argument e and returns true when all samples are true). Our proofs of these are exactly analgous to those described by Marionneau et al. [42] and we will not replicate them here, however we highlight that it is important that this program is higher-order, as the Gaussian sampler dynamically chooses the parameters to G and I. Using these combinators we can implement Z, Karney's sampler for the integer part of the Gaussian distribution.[7]

The program itself is a simple rejection sampling loop, and involves no real numbers beyond those used internally by H. Proving that Z is distributed as μ_{Z} amounts to applying the specifications we have developed so far. At a high level, we begin with $\lightning(V)$ credits where $V \triangleq \sum_{k=0}^{\infty} \mu_{\mathsf{Z}}(k) \cdot F(k)$ for some $F \in \mathcal{B}(\mathbb{N})$. Like before, the proof proceeds by Löb induction, conditioning the credits on the results of G and I by g and h respectively:

$$
\begin{aligned}
h(k,b) &\triangleq [b] \cdot F(k) + [\neg b] \cdot V \\
g(k) &\triangleq e^{-k(k-1)/2} h(k, \mathsf{true}) + (1 - e^{-k(k-1)/2}) h(k, \mathsf{false})
\end{aligned}
$$

The fact that their expected value does not exceed the initial amount of error credits follows from Fubini's theorem.

4.3 Gaussian Sampler

A RandML implementation of Karney's sampler for the Gaussian distibution is given in N′ in Figure 9. More precisely, the N′ sampler draws from the *half* or *one-sided* Gaussian distribution over $[0, \infty)$, represented as a pair containing a nonnegative integer and a lazy

[7] This sampler corresponds to steps `N1` and `N2` from Karney's paper.

uniform real. The full Gaussian sampler N transforms a sample from N' into the full normal distribution over $(-\infty, \infty)$ in N by choosing a sign uniformly at random, and applying RofBZU to combine our samples into our constructive real library from §2.5.

The implementation of N' involes a complicated rejection step B, which (like Karney) we split into a number of steps:

- $\mathsf{C}\ M$: returns -1, 0, and 1 with probabilities $1/M$, $1/M$, and $(M-2)/M$ respectively, when $1 \le M$.
- $\mathsf{A}\ k\ x$: A bernoulli trial with parameter $(1-(2k+x)/(2k+2))$.
- $\mathsf{S}'\ k\ x$: A $\mathcal{D}(\mathbb{N})$ sampler with PMF

$$\frac{x^n}{n!}\left(\frac{2k+x}{2k+2}\right)^n - \frac{x^{n+1}}{(n+1)}\left(\frac{2k+x}{2k+2}\right)^{n+1}$$

  S is a sampling loop for this program, written in tail-recursive form. This program is similar to the loop in R, but with additional *thinning* introduced by A.
- $\mathsf{B}\ k\ x$: A $e^{-x(2k+x)/(2k+2)}$ Bernoulli trial, based on Von-Neumann's technique.

Aside from the spike in complexity, the techniques we have developed so far are capable of verifying this composition of samplers with no substantial modifications. In our Rocq development we prove specifications for each helper function in the style of (3), quantifying over functions in $\mathcal{B}(\{-1,0,1\})$, $\mathcal{B}(\{\mathsf{true},\mathsf{false}\})$, or $\mathcal{B}(\mathbb{N})$, which are used modularly to condition error credits in their clients. Each helper function is a discrete sampler, though some of them (like S) make internal use of real numbers to sample discrete values with the correct probabilities. The Appendix [16] contains the credit distribution functions for each of these helpers, as well as the proof that their expected value does not exceed the initial supply of error credits. The most challenging aspect of our approach is determining a closed form for the distribution of each helper function, which is necessary to state specifications in the form of (3).

With B in hand, we can verify the correctness of the one-sided Gaussian sampler N'. Because the return value for this program is a pair of a (nonnegative) integer and a lazy real, our specification expresses the density of N' as

$$\mu_{\mathsf{N}'}(n,x) \triangleq e^{-(n+x)^2/2}/\mathcal{N}_{\mathsf{N}'} \qquad \mathcal{N}_{\mathsf{N}'} \triangleq \int_0^1 \sum_{k=0}^{\infty} e^{-(k+x)^2/2}\,dx \tag{22}$$

Finally, we define the symmetric normal distribution N by lifting our Gaussian samples to a CReal, choosing the sign uniformly at random. For any bounded, piecewise continuous $F : \mathbb{R} \to \mathbb{R}^{\ge 0}$ we can show that

$$\left\{ \lightning\left(\int_{-\infty}^{\infty} \frac{e^{-x^2/2}}{2\mathcal{N}_{\mathsf{N}'}} \cdot F(x)\,dx\right)\right\}\ \mathsf{N}\ ()\ \{v.\ \exists r.\ \mathrm{IsApprox}\ v\ r * \lightning(F(r))\}. \tag{23}$$

To prove this, we instantiate the specification for N' with the function $G(n,x) = F(-(n+x)) + F(n+x)$, and then use the final rand 1 step to distribute that credit to the positive and negative cases appropriately. Our specification for RofBZU performs the final task of lifting this result to a CReal. The specification (23) is now in a form for which we can apply the adequacy theorems, and this completes our verification.

```
E N ≜ let x := U in
      let y := R 0 x in
      if y % 2 = 0 then (x, N) else E (N + 1)
```

Figure 10 The Negative Exponential sampler (E).

```
L ε μ ≜ let (z, u) := E 0 in
        let sgn := rand 1 in
        Radd μ (Rscal ε (RofBZU (sgn, z, u)))
```

Figure 11 The Laplace sampler (L).

5 Verified Laplace Sampler

In this section, we verify a sampler for the continuous Laplace distribution $\mathcal{L}_{\varepsilon,\mu}$, which has important applications in differentially private algorithms. Its density function is given by:

$$\mathcal{L}_{\varepsilon,\mu}(x) = \frac{\varepsilon}{2} \cdot e^{-\varepsilon \cdot |x-\mu|} \tag{24}$$

Our starting point is a sampler for the negative exponential distribution (§5.1). Samples from this distribution can be transformed into Laplace samples by applying appropriate arithmetic operations, which we implement using our verified library for constructive real arithmetic (§2.5). Finally, we prove a tail bound on the size of Laplace samples, which has applications for proving *accuracy* bounds of differentially private algorithms (§5.2).

5.1 Laplace Sampling

The Laplace sampler uses the sampler E for the negative exponential distribution shown in Figure 10. Karney [39] calls this *Algorithm V*. Similarly to the one-sided Gaussian sampler, E returns pairs in $\mathbb{N} \times [0, 1]$. In this case, the probability of returning (z, x) is $\mu_{\mathsf{E}} \triangleq e^{-(z+x)}$. The proof of this fact is similar enough to the proof of the Gaussian sampler in §4 that we will omit it and direct interested readers to the appendix or Rocq development.

We must make three modifications to the exponential mechanism in order to turn it into a sampler for the Laplace distribution:

1. Extend the distribution by symmetry to all of $\mathbb{R}$,
2. Multiply the result by a scaling factor ε, and
3. Shift the distribution by the mean μ.

We can implement all three of these steps using our constructive reals library as described in §2.5. Figure 11 depicts the implementation of our Laplace sampler. First, after taking a sample from E, one can decide the sign of the result using a rand 1 sample. The boolean-integer-uniform triple is lifted into a single constructive real value using the RofBZU function which, like the Gaussian, gives us a sample from the symmetric version of the negative exponential distribution. Now we are able to perform arithmetic on this sample: we can scale and shift the returned value using our constructive reals library.

After obtaining the sample from the negative exponential function and symmetrizing it like we did in §4, the remaining operations are all deterministic. Putting everything together, we can prove that for any $F \in \mathcal{PC}(\mathbb{R})$,

$$\left\{ \lightning \left(\int_{-\infty}^{\infty} F(x) \mathcal{L}_{2^\varepsilon,\mu}\, dx \right) * \text{IsApprox } v_\mu\ \mu\ I_\mu \right\} \sqsubseteq \varepsilon\ v_\mu \left\{ v.\, \exists I_L, r. \begin{array}{l} \lightning (F\ r) * I_L * \\ \text{IsApprox } v\ r\ (I_\mu * I_L) \end{array} \right\} \tag{25}$$

5.2 Accuracy Bound

As mentioned previously, the continuous Laplace distribution is widely used in differential privacy applications. With differential privacy, random noise is added to the results of computations to avoid leaking private information about the data used in the computation. Adding more noise gives stronger privacy guarantees, but adding too much noise can make the computed value less useful. Thus, an important consideration in analyzing differentially private algorithms is to consider the *utility* or *accuracy* of the algorithm. For example, in defining the *Report Noisy Max* algorithm, Dwork and Roth [18] establish a utility bound on a differentially private query program based on the following property of the Laplace distribution:

$$\mathbb{P}\left[|\mathcal{L}_{\varepsilon,\mu} - \mu| > \frac{\log(1/\beta)}{\varepsilon} \right] \leq \beta \tag{26}$$

Barthe et al. [4] incorporated a similar bound for the *discrete* Laplace distribution as a rule in aHL, a program logic for reasoning about error bounds, where sampling from the discrete Laplace was added as a primitive to the language.

Instead of adding this bound as a primitive rule in our logic, our specification for the continuous Laplace sampler allows us to derive this bound for our sampler implementation in a straightforward way. First, we instantiate the specification (25) with F being the indicator function for the set

$$S \triangleq \left(-\infty, \mu - \frac{\log(1/\beta)}{2^\varepsilon} \right] \cup \left[\mu + \frac{\log(1/\beta)}{2^\varepsilon}, \infty \right)$$

Then, computing the improper integral in (25), we can use the fact that $F(x) = 1$ for $x \in S$ together with the *spending* rule to prove the following Continuous-Eris equivalent of the approximate specification (26):

$$\{ \lightning(\beta) * \text{IsApprox } v_\mu\ \mu\ I_\mu \} \sqsubseteq \varepsilon\ v_\mu \left\{ f.\, \exists I_L, r. \begin{array}{l} (|r - \mu| < \frac{\log(1/\beta)}{2^\varepsilon}) * I_L * \\ \text{IsApprox } v\ r\ (I_\mu * I_L) \end{array} \right\}$$

6 Related Work

We have already alluded to the extensive literature on different representations of computable reals and operations on them. The implementation we have verified is a simplified adaptation of the CReal library [22] which is based on the work of Ménissier-Morain [43]. The actual CReal library implementation uses additional mutable state to cache approximations of numbers for efficiency. Since our language includes mutable state, it should be possible to generalize our verification to handle this as well.

SampCert [17] verifies a number of discrete samplers in the Lean theorem prover. Unlike Continuous-Eris, which supports the verification of programs that use higher-order random functions and state (features known to be challenging to reason about denotationally),

SampCert's object language is restricted to first-order programs. Hence, it cannot support samplers based on lazy reals. The SampCert development includes a verified implementation of the discrete Gaussian sampler, a two-sided version of our sampler Z from §4.2 which can have variance $\sigma \in \mathbb{Q}$ and mean $\mu \in \mathbb{Z}$.

Zar [3] is a formally verified compiler for discrete probabilistic programs that generates executable samplers. Its proof shows that the generated code produces samples with the correct probability distribution.

Although we have focused on two exact samplers for continuous distributions, a range of algorithms exist for different distributions and stochastic processes. Huber [35] and Occil [47] provide extensive overviews of these algorithms. A related line of work explores what probability distributions are computable, *i.e.* in the sense that one can computably approximate the value of the measure or density defining a distribution [23, 1, 33, 19, 9].

As alluded to earlier, the semantics of languages that have sampling from continuous probability distributions as a primitive poses some technical challenges, particularly when combined with other semantically rich language features, such as higher-order functions and general recursion. A number of approaches to the denotational semantics of higher-order probabilistic programs that feature continuous sampling have been proposed [30, 20, 13, 52, 36]. Others have explored operational semantics for such languages [12] and logical relations techniques for proving equivalences [55]. Because the language we consider only has discrete probabilistic sampling as a primitive, and uses this to implement computable versions of continuous sampling, we avoid the semantic challenges of continuous distributions, while meanwhile still handling other features such as mutable higher-order state. Huang and Morrisett [34] used computable distributions as the basis for a denotational semantics of a probabilistic programming language, and gave a sampler implementation for this approach.

Dash et al. [14] develop a monadic probabilistic programming language with support for laziness, and show how to use it to express various stochastic processes, such as Poisson and Gaussian processes. Since these processes are an infinite collection of random variables, they cannot be sampled in their entirety. Instead, a key idea is to lazily sample parts of a process and then cache the sampled values, similar to how the U sampler we considered lazily samples and stores the bits of a uniform $[0, 1]$ sample. Their language supports a form of conditioning, and can be used for Bayesian modeling with a Monte Carlo inference algorithm. They give a denotational semantics in terms of quasi-Borel spaces and an implementation called LazyPPL in Haskell. LazyPPL assumes the existence of primitives for sampling directly from continuous distributions such as the Gaussian, which are implemented in practice using floats. In contrast, our focus has been on proving the correctness of computable samplers for these continuous distributions, and we have used an operational semantics.

Recently, a number of program logics [40, 32, 49, 50, 31] have been developed for reasoning about programs that sample from continuous probability distributions. These logics all treat sampling from continuous distributions and exact real operations as primitives, as opposed to our approach of verifying computable implementations of these operations. Moreover, these logics do not support mutable state or dynamically allocated memory. In contrast, Continuous-Eris inherits the Iris framework's support for reasoning about state and pointers. On the other hand, some of these logics support language constructs for conditioning on observable events, as needed for Bayesian probabilistic modeling, which Continuous-Eris does not handle.

The proof outlined in §2.3 for generalizing Eris's discrete error credit rules into continuous versions exploits the fact that the Riemann integral can be approximated by Riemann sums. Batz et al. [5] have also used Riemann sums to approximate integrals in verifying probabilistic

programs. They observe that when using a pre-expectation calculus for a program with continuous samples, the standard approach gives rise to terms involving integrals, which poses a challenge for automated verification. They show that by instead replacing these integrals with approximations by Riemann sums, one obtains a version that is more amenable to automation, while still yielding sound bounds on the original problem. Beutner et al. [7] similarly use Riemann sums in bounding properties of probabilistic programs.

Although it is common to use floating-point approximations of continuous distributions, Garg et al. [24] instead consider using *fixed-point* approximations in a probabilistic programming language for Bayesian inference. They show that it is then possible to apply exact techniques for discrete Bayesian inference to this approximation, while soundly controlling the errors introduced by discretization.

7 Conclusion

Exact sampling algorithms for continuous distributions involve a combination of features that are challenging to reason about. Continuous-Eris provides a way to verify these algorithms using infinite presampling tapes, which are derived by extending Eris with time receipts.

Future Work

There are other logics related to Eris that also feature presampling tapes and mechanisms that behave similarly to error credits, such as Approxis [28], which allows for relational reasoning, and Tachis [29], for bounding expected costs. It would be interesting to also extend these logics with time receipts and derive infinite presampling tapes to be able to apply them to programs that use exact samplers from continuous distributions.

Another direction for future work is to address the restrictions associated to Riemann integration. One concrete limitation of the current approach is that our credit distribution functions are required to be piecewise continuous, and therefore cannot express credit distributions which have an uncountable number of discontinuities. Integrability notwithstanding, one might imagine using the indicator function on the rationals as a credit distribution function to prove that the uniform sampler avoids all rational points. In its current form proving this in Continuous-Eris requires one to apply Theorem 4, thereby exiting the program logic and giving up the ability to use this fact as part of a larger Continuous-Eris proof. Generalizations to more permissive integrals such as the Lebesgue or Henstock–Kurzweil integral may alleviate these side conditions.

References

1 Nathanael L. Ackerman, Cameron E. Freer, and Daniel M. Roy. On the computability of conditional probability. *J. ACM*, 66(3):23:1–23:40, 2019. doi:10.1145/3321699.

2 Alejandro Aguirre, Philipp G. Haselwarter, Markus de Medeiros, Kwing Hei Li, Simon Oddershede Gregersen, Joseph Tassarotti, and Lars Birkedal. Error credits: Resourceful reasoning about error bounds for higher-order probabilistic programs. *Proc. ACM Program. Lang.*, 2024. doi:10.1145/3674635.

3 Alexander Bagnall, Gordon Stewart, and Anindya Banerjee. Formally verified samplers from probabilistic programs with loops and conditioning. *Proc. ACM Program. Lang.*, 7(PLDI):1–24, 2023. doi:10.1145/3591220.

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

5 Kevin Batz, Joost-Pieter Katoen, Francesca Randone, and Tobias Winkler. Foundations for deductive verification of continuous probabilistic programs: From lebesgue to riemann and back. *Proc. ACM Program. Lang.*, 9(OOPSLA1):421–448, 2025. doi:10.1145/3720429.

6 Andrej Bauer and Iztok Kavkler. Implementing real numbers with RZ. In Ruth Dillhage, Tanja Grubba, Andrea Sorbi, Klaus Weihrauch, and Ning Zhong, editors, *Proceedings of the Fourth International Conference on Computability and Complexity in Analysis, CCA 2007, Siena, Italy, June 16-18, 2007*, volume 202 of *Electronic Notes in Theoretical Computer Science*, pages 365–384. Elsevier, 2007. doi:10.1016/J.ENTCS.2008.03.027.

7 Raven Beutner, C.-H. Luke Ong, and Fabian Zaiser. Guaranteed bounds for posterior inference in universal probabilistic programming. In Ranjit Jhala and Isil Dillig, editors, *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13–17, 2022*, pages 536–551. ACM, 2022. doi:10.1145/3519939.3523721.

8 Patrick Billingsley. *Probability and Measure*. Wiley, 3rd edition, 1995.

9 Paul Bilokon and Abbas Edalat. A domain-theoretic approach to brownian motion and general continuous stochastic processes. In Thomas A. Henzinger and Dale Miller, editors, *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS 2014, Vienna, Austria, July 14–18, 2014*, pages 15:1–15:10. ACM, 2014. doi:10.1145/2603088.2603102.

10 Hans-Juergen Boehm. Towards an API for the real numbers. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 562–576. ACM, 2020. doi:10.1145/3385412.3386037.

11 Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. Coquelicot: A user-friendly library of real analysis for Coq. *Math. Comput. Sci.*, 9(1):41–62, 2015. doi:10.1007/S11786-014-0181-1.

12 Johannes Borgström, Ugo Dal Lago, Andrew D. Gordon, and Marcin Szymczak. A lambda-calculus foundation for universal probabilistic programming. In Jacques Garrigue, Gabriele Keller, and Eijiro Sumii, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 33–46. ACM, 2016. doi:10.1145/2951913.2951942.

13 Fredrik Dahlqvist and Dexter Kozen. Semantics of higher-order probabilistic programs with conditioning. *Proc. ACM Program. Lang.*, 4(POPL):57:1–57:29, 2020. doi:10.1145/3371125.

14 Swaraj Dash, Younesse Kaddar, Hugo Paquet, and Sam Staton. Affine monads and lazy structures for bayesian programming. *Proc. ACM Program. Lang.*, 7(POPL):1338–1368, 2023. doi:10.1145/3571239.

15 Markus de Medeiros, Puming Liu, Kwing Hei Li, Alejandro Aguirre, Lars Birkedal, and Joseph Tassarotti. Artifact for "Verifying exact samplers for continuous distributions with a discrete program logic", May 2026. doi:10.5281/zenodo.20114732.

16 Markus de Medeiros, Puming Liu, Kwing Hei Li, Alejandro Aguirre, Lars Birkedal, and Joseph Tassarotti. Verifying exact samplers for continuous distributions with a discrete program logic, 2026. arXiv:2605.13526.

17 Markus de Medeiros, Muhammad Naveed, Tancrède Lepoint, Temesghen Kahsai, Tristan Ravitch, Stefan Zetzsche, Anjali Joshi, Joseph Tassarotti, Aws Albarghouthi, and Jean-Baptiste Tristan. Verified foundations for differential privacy. *Proc. ACM Program. Lang.*, 9(PLDI), 2025. doi:10.1145/3729294.

18 Cynthia Dwork and Aaron Roth. The algorithmic foundations of differential privacy. *Found. Trends Theor. Comput. Sci.*, 9(3–4):211–407, 2014. doi:10.1561/0400000042.

19 Abbas Edalat. A computable approach to measure and integration theory. *Inf. Comput.*, 207(5):642–659, 2009. doi:10.1016/J.IC.2008.05.003.

20 Thomas Ehrhard, Michele Pagani, and Christine Tasson. Measurable cones and stable, measurable functions: A model for probabilistic higher-order programming. *Proc. ACM Program. Lang.*, 2(POPL):59:1–59:28, 2018. doi:10.1145/3158147.

21 Martín Hötzel Escardó. PCF extended with real numbers. *Theor. Comput. Sci.*, 162(1):79–115, 1996. doi:10.1016/0304-3975(95)00250-2.

22 Jean-Christophe Filliâtre. Creal library. URL: https://usr.lmf.cnrs.fr/~jcf/creal.en.html.

23 Cameron E. Freer and Daniel M. Roy. Computable de finetti measures. *Ann. Pure Appl. Log.*, 163(5):530–546, 2012. doi:10.1016/J.APAL.2011.06.011.

24 Poorva Garg, Steven Holtzen, Guy Van den Broeck, and Todd D. Millstein. Bit blasting probabilistic programs. *Proc. ACM Program. Lang.*, 8(PLDI):865–888, 2024. doi:10.1145/3656412.

25 Pietro Di Gianantonio. Real number computability and domain theory. *Inf. Comput.*, 127(1):11–25, 1996. doi:10.1006/INCO.1996.0046.

26 Michèle Giry. A categorical approach to probability theory. In B. Banaschewski, editor, *Categorical Aspects of Topology and Analysis*, pages 68–85, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg.

27 Simon Oddershede Gregersen, Alejandro Aguirre, Philipp G. Haselwarter, Joseph Tassarotti, and Lars Birkedal. Asynchronous probabilistic couplings in higher-order separation logic. *Proc. ACM Program. Lang.*, 8(POPL):753–784, 2024. doi:10.1145/3632868.

28 Philipp G. Haselwarter, Kwing Hei Li, Alejandro Aguirre, Simon Oddershede Gregersen, Joseph Tassarotti, and Lars Birkedal. Approximate relational reasoning for higher-order probabilistic programs. *Proc. ACM Program. Lang.*, 9(POPL):1196–1226, 2025. doi:10.1145/3704877.

29 Philipp G. Haselwarter, Kwing Hei Li, Markus de Medeiros, Simon Oddershede Gregersen, Alejandro Aguirre, Joseph Tassarotti, and Lars Birkedal. Tachis: Higher-order separation logic with credits for expected costs. *Proc. ACM Program. Lang.*, 8(OOPSLA2):1189–1218, 2024. doi:10.1145/3689753.

30 Chris Heunen, Ohad Kammar, Sam Staton, and Hongseok Yang. A convenient category for higher-order probability theory. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*, pages 1–12. IEEE Computer Society, 2017. doi:10.1109/LICS.2017.8005137.

31 Michikazu Hirata, Yasuhiko Minamide, and Tetsuya Sato. Program logic for higher-order probabilistic programs in isabelle/hol. In Michael Hanus and Atsushi Igarashi, editors, *Functional and Logic Programming – 16th International Symposium, FLOPS 2022, Kyoto, Japan, May 10-12, 2022, Proceedings*, volume 13215 of *Lecture Notes in Computer Science*, pages 57–74. Springer, 2022. doi:10.1007/978-3-030-99461-7_4.

32 Shing Hin Ho, Nicolas Wu, and Azalea Raad. Bayesian separation logic: A logical foundation and axiomatic semantics for probabilistic programming. *Proc. ACM Program. Lang.*, 10(POPL), 2026. doi:10.1145/3776696.

33 Mathieu Hoyrup and Cristóbal Rojas. Computability of probability measures and martin-löf randomness over metric spaces. *Information and Computation*, 207(7):830–847, 2009. doi:10.1016/j.ic.2008.12.009.

34 Daniel Huang and Greg Morrisett. An application of computable distributions to the semantics of probabilistic programming languages. In Peter Thiemann, editor, *Programming Languages and Systems – 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9632 of *Lecture Notes in Computer Science*, pages 337–363. Springer, 2016. doi:10.1007/978-3-662-49498-1_14.

35 M.L. Huber. *Perfect Simulation*. Chapman & Hall/CRC Monographs on Statistics & Applied Probability. CRC Press, 2016.

36 Mathieu Huot, Alexander K. Lew, Vikash K. Mansinghka, and Sam Staton. ωpap spaces: Reasoning denotationally about higher-order, recursive probabilistic and differentiable programs. In *38th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2023, Boston, MA, USA, June 26-29, 2023*, pages 1–14. IEEE, 2023. doi:10.1109/LICS56636.2023.10175739.

37 Vernon A. Lee Jr. and Hans-Juergen Boehm. Optimizing programs over the constructive reals. In Bernard N. Fischer, editor, *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI), White Plains, New York, USA, June 20-22, 1990*, pages 102–111. ACM, 1990. doi:10.1145/93542.93558.

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

39 Charles F. F. Karney. Sampling exactly from the normal distribution. *ACM Trans. Math. Softw.*, 42(1), 2016. doi:10.1145/2710016.

40 John M. Li, Amal Ahmed, and Steven Holtzen. Lilac: A modal separation logic for conditional probability. *Proc. ACM Program. Lang.*, 7(PLDI):148–171, 2023. doi:10.1145/3591226.

41 Kwing Hei Li, Alejandro Aguirre, Simon Oddershede Gregersen, Philipp G. Haselwarter, Joseph Tassarotti, and Lars Birkedal. Modular reasoning about error bounds for concurrent probabilistic programs. *Proc. ACM Program. Lang.*, 9(ICFP), 2025. doi:10.1145/3747514.

42 Virgil Marionneau, Félix Sassus Bourda, Alejandro Aguirre, and Lars Birkedal. 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*, CPP '26, pages 368–382, New York, NY, USA, 2026. Association for Computing Machinery. doi:10.1145/3779031.3779109.

43 Valérie Ménissier-Morain. Arbitrary precision real arithmetic: design and algorithms. Research Report lip6.2003.003, LIP6, May 2003. URL: https://hal.science/hal-02545650.

44 Glen Mével, Jacques-Henri Jourdan, and François Pottier. Time Credits and Time Receipts in Iris. In *Lecture Notes in Computer Science*, volume 11423 of *Lecture Notes in Computer Science*, pages 3–29, Prague, Czech Republic, April 2019. Springer. doi:10.1007/978-3-030-17184-1_1.

45 Ilya Mironov. On significance of the least significant bits for differential privacy. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 650–661, 2012. doi:10.1145/2382196.2382264.

46 Norbert Th. Müller. The iRRAM: Exact arithmetic in C++. In Jens Blanck, Vasco Brattka, and Peter Hertling, editors, *Computability and Complexity in Analysis, 4th International Workshop, CCA 2000, Swansea, UK, September 17-19, 2000, Selected Papers*, volume 2064 of *Lecture Notes in Computer Science*, pages 222–252. Springer, 2000. doi:10.1007/3-540-45335-0_14.

47 Peter Occil. Partially-sampled random numbers for accurate sampling of continuous distributions. URL: https://peteroupc.github.io/exporand.pdf.

48 Peter John Potts, Abbas Edalat, and Martín Hötzel Escardó. Semantics of exact real arithmetic. In *Proceedings, 12th Annual IEEE Symposium on Logic in Computer Science, Warsaw, Poland, June 29 – July 2, 1997*, pages 248–257. IEEE Computer Society, 1997. doi:10.1109/LICS.1997.614952.

49 Tetsuya Sato. Approximate relational hoare logic for continuous random samplings. In Lars Birkedal, editor, *The Thirty-second Conference on the Mathematical Foundations of Programming Semantics, MFPS 2016, Carnegie Mellon University, Pittsburgh, PA, USA, May 23-26, 2016*, volume 325 of *Electronic Notes in Theoretical Computer Science*, pages 277–298. Elsevier, 2016. doi:10.1016/J.ENTCS.2016.09.043.

50 Tetsuya Sato, Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Justin Hsu. Formal verification of higher-order probabilistic programs: reasoning about approximation, convergence, bayesian inference, and optimization. *Proc. ACM Program. Lang.*, 3(POPL):38:1–38:30, 2019. doi:10.1145/3290351.

51 Alex K. Simpson. Lazy functional algorithms for exact real functionals. In Lubos Brim, Jozef Gruska, and Jirí Zlatuska, editors, *Mathematical Foundations of Computer Science 1998, 23rd International Symposium, MFCS'98, Brno, Czech Republic, August 24-28, 1998, Proceedings*, volume 1450 of *Lecture Notes in Computer Science*, pages 456–464. Springer, 1998. doi:10.1007/BFB0055795.

52 Matthijs Vákár, Ohad Kammar, and Sam Staton. A domain theory for statistical probabilistic programming. *Proc. ACM Program. Lang.*, 3(POPL):36:1–36:29, 2019. doi:10.1145/3290349.

53 J. von Neumann. Various techniques used in connection with random digits. In A. S. Householder, G. E. Forsythe, and H. H. Germond, editors, *Monte Carlo Method*, pages 36–38, (NBS, Washington, DC), proceedings of a symposium held June 29–July 1, 1949, in Los Angeles, 1951.

54 Jean Vuillemin. Exact real computer arithmetic with continued fractions. *IEEE Trans. Computers*, 39(8):1087–1105, 1990. doi:10.1109/12.57047.

55 Mitchell Wand, Ryan Culpepper, Theophilos Giannakopoulos, and Andrew Cobb. Contextual equivalence for a probabilistic language with continuous random variables and recursion. *Proc. ACM Program. Lang.*, 2(ICFP):87:1–87:30, 2018. doi:10.1145/3236782.

56 E. Wiedmer. Computing with infinite objects. *Theoretical Computer Science*, 10:133–155, 1980. doi:10.1016/0304-3975(80)90011-0.