Type-Directed Determinization of Higher-Order Probabilistic Programs

ANONYMOUS AUTHOR(S)

Probabilistic programs often use continuous random variables, and their expected output is then estimated by sampling. In this paper we explore the insight that some of these random samples can be replaced by their means without changing the expected output, thereby reducing the variance of the estimate. We design *determinization*, a type-directed, source-to-source transformation that automatically identifies and replaces such samples in higher-order, recursive probabilistic programs. Our type system tracks dependencies between random choices and restricts determinizable values to affine uses. This makes the replacements sound without symbolic marginalization. Using a symbolic operational semantics, we prove that determinization preserves expected values and never increases output variance, and mechanize these guarantees in Lean. Our evaluation shows that determinization reduces output variance on sampling benchmarks from prior work and on new examples, in some cases by more than two orders of magnitude. When determinization eliminates all continuous random variables, it also allows exact computation of expectations, including for programs that existing exact inference tools do not support.

1 Introduction

Probabilistic programming languages (PPLs) extend standard languages with constructs for manipulating random variables [12, 14, 24, 30, 38, 53]. Many PPLs also support conditionals, higher-order functions, and recursion, which make programs expressive but harder to analyze. A basic question is what a probabilistic program returns on average, that is, the expected value of its output.

Expectations can be computed exactly, using symbolic methods [28, 29, 45], or estimated like in Monte Carlo methods, by running the program many times. Hybrid approaches, such as conditional Monte Carlo and Rao–Blackwellization, average over some random choices analytically rather than sampling them to reduce variance [15, 48]. One can also achieve reduced variance by interleaving symbolic reasoning with sampling during inference [2, 46].

The core question we address in our paper is motivated by these methods: which random choices can we remove from the program without changing its expected value? One can quickly see that simply replacing a sample by its mean need not preserve this value. The sample may determine a branch, set the parameter of a later distribution, or enter a nonlinear computation, all of which might affect the soundness of the replacement. We will overcome these challenges by using types!

To make the challenges more concrete, consider the program on the right, which samples a signal x uniformly from $[0, 1]$, then samples a Gaussian measurement y with mean x and a Gaussian measurement z with mean y, then returns the product xz. Once x is fixed, the measurement noise contributes nothing to the expected output:

```
let x = uniform(0, 1) in
let y = gaussian(x, 1) in
let z = gaussian(y, 1) in
x * z
```

$$\mathbb{E}[xz \mid x] = x\,\mathbb{E}[z \mid x] = x\,\mathbb{E}[\mathbb{E}[z \mid x, y] \mid x] = x\,\mathbb{E}[y \mid x] = x^2.$$

Replacing both Gaussian samples by their means yields a simpler program that samples only x and returns x^2: `let x = uniform(0, 1) in x * x`. Both the source and transformed programs have expected value $1/3$, but replacing the measurement noise reduces output variance from $\frac{34}{45}$ to $\frac{4}{45}$, a factor of $8.50\times$.

This example also exposes the central difficulty in the transformation: had we also replaced the uniform sample by its mean, $1/2$, the resulting program would return $1/4$ rather than $1/3$ and hence not be equivalent to the source program. Thus, only two random variables can be soundly

2026. ACM XXXX-XXXX/2026/10-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

replaced in this example, and the third must remain. Whether a sample can be replaced depends on how its value is used: the returned value of the above program is linear in z when x is fixed, but becomes quadratic in x after averaging over y and z. A sound transformation must therefore reason about both the dependencies between random choices and their uses throughout the program. This challenge becomes particularly pronounced in higher-order, potentially unbounded recursive programs, where the influence of a random choice can propagate through function calls and across arbitrarily long executions.

Our key insight is that safe replacements can nevertheless be identified compositionally from how sampled values are used, without requiring symbolic marginalization of their distributions. We introduce *determinization*, a type-directed, source-to-source transformation of higher-order, recursive probabilistic programs with continuous distributions that automatically identifies which samples can be replaced by their means.

Our type system will distinguish sampling choices that remain random (annotated by G) from those that can be eliminated (E). Revisiting the signal example from earlier would yield the type annotated program on the right. Type inference automatically determines these modes by tracking how sampled values are used: values marked E may participate in affine computations, but cannot determine control flow or the parameters of retained samples.

```
let x = uniform[G](0, 1) in
let y = gaussian[E](x, 1) in
let z = gaussian[E](y, 1) in
x * z
```

Our transformation then replaces each E-sample by its mean. We establish soundness of this type-directed transformation through a relational proof that connects the original and transformed programs via two interpretations of a common symbolic semantics. The resulting program preserves the expected value of the output while never increasing its variance.

The next three examples illustrate, at a high level, how determinization handles higher-order functions, data structures, and unbounded recursion.

Consider an example of a Gaussian random walk in Fig. 1 (on the left). This *higher-order* program passes a probabilistic step function to `reduce`, a left fold that traverses the requested times and accumulates *a list* of time-position pairs. At each time t', it samples a new position y' from $\mathcal{N}(y, t' - t)$, where (t, y) is the latest pair in the path. It then adds (t', y') to the front of the path. At the next step, it reads that pair and uses y' as the next Gaussian's mean. Our type inference procedure will mark the Gaussian sampling step as determinizable without unrolling the traversal. Replacing the draws by their means keeps every position at zero, yielding a deterministic path that matches the expected position at each requested time.

```
let crawl = fun loot =>
  (rec visit d =>
    if flip[G](0.25) then 0
    else loot d + visit (d + 1)
  ) 1
in
crawl (fun d => 200 * d
              * bernoulli[E](0.01))
```

For recursion, consider the dungeon crawl program on the left, which recursively visits rooms until an exit coin ends the computation, with no fixed bound on the number of rooms visited. At depth d, a room pays a jackpot of $200d$ with probability 0.01. The type inference will identify that the exit coins remain random because they control recursion, while each jackpot contributes linearly to the total reward and can be replaced by its mean payout, $2d$. The determinized program will have the last two lines replaced by `crawl (fun depth => 2 * depth)` and one can show that it has the same expected value. Notably, once again this type-directed transformation reduces output variance (this time by a factor of 12.36), with the new program needing correspondingly fewer samples to achieve the same estimator variance.

Our determinization procedure can in some cases go beyond improving sampling: it can enable exact expectation computation by transforming an infinite-state program into a finite-state one.

Source (annotations inferred)

```
let step = fun path => fun t' =>
  match path with
    | [] => []
    | current :: _ =>
        let t = fst current in
        let y = snd current in
        let y' = gaussian[E](y, t' - t) in
        (t', y') :: path
in
let times =
  1 :: 2 :: 4 :: 5.5 :: 6 :: 6.25 :: []
in
reduce step ((0, 0) :: []) times
```

After replacement (simplified)

```
let step = fun path => fun t' =>
  match path with
    | [] => []
    | current :: _ =>
        let y = snd current in


        (t', y) :: path
in
let times =
  1 :: 2 :: 4 :: 5.5 :: 6 :: 6.25 :: []
in
reduce step ((0, 0) :: []) times
```

Fig. 1. Gaussian random walk from [52], consisting of a higher order function reduce and a recursive traversal of lists of pairs. Full program in Section B.

```
(rec run x =>
  if flip[G](0.5) then x
  else run
   (1 - x + gaussian[E](0, 1))
) 0
```

Consider, on the left, a recursive process that either returns its current state x or continues from value $1 - x$ plus Gaussian noise, each choice with probability $1/2$. Type inference labels the two random variables adequately and enables determinization to identify that the gaussian can be replaced by its mean.

Hence determinization removes the noise, yielding the program on the right: starting from zero, the transformed loop has only two possible states, 0 and 1. Let v_i denote the expected value eventually returned by the transformed program when entering the loop with $\mathtt{x} = i$, accounting for all subsequent iterations. The expected outputs from these two states satisfy

```
(rec run x =>
  if flip[G](0.5) then x
  else run (1 - x)
) 0
```

$$\begin{pmatrix} v_0 \\ v_1 \end{pmatrix} = \begin{pmatrix} 0 & \frac{1}{2} \\ \frac{1}{2} & 0 \end{pmatrix} \begin{pmatrix} v_0 \\ v_1 \end{pmatrix} + \begin{pmatrix} 0 \\ \frac{1}{2} \end{pmatrix} \qquad \Longrightarrow \qquad \begin{pmatrix} v_0 \\ v_1 \end{pmatrix} = \begin{pmatrix} \frac{1}{3} \\ \frac{2}{3} \end{pmatrix}.$$

Since the program starts with $\mathtt{x} = 0$, its expected output is $v_0 = \frac{1}{3}$. A continuous-state computation thus becomes a finite-state problem whose expectation can be computed exactly. Our implementation produces a Lean-checkable certificate for the resulting exact value.

In a nutshell, our paper makes the following contributions:

Type system and inference. We develop a type system and source-to-source transformation that identifies and replaces eligible samples by their means, even in higher-order, potentially unbounded recursive programs. Our inference algorithm finds the greatest assignment of determinizable sites permitted by the type system, without requiring symbolic marginalization or unrolling recursive computations (Sections 2 and 3).

Soundness. We prove that determinization preserves expected values and never increases output variance, under suitable assumptions. Since the transformation changes the output distribution, its correctness requires relating the original and transformed executions rather than showing distributional equivalence. We develop a relational proof based on two interpretations of a common symbolic semantics, establishing a stronger tracewise soundness result from which the global guarantees follow (Sections 4 and 5).

Lean implementation and verification. We implement type inference, determinization, and sampling in Lean, and mechanize our soundness theorems. Our implementation additionally supports verified matrix solving and certificate checking, providing an end-to-end correctness guarantee: if the original program is domain-safe and its expectation is well-defined, the computed expectation is exactly that of the original program (Section 6). The Lean code is included in the artifact. References to mechanized definitions and theorems carry a superscript Lean logo, as in [Lean].

Evaluation. We evaluate two applications of determinization. First, by eliminating continuous sampling, determinization can reduce infinite-state programs to finite-state programs, enabling exact computation of their expected values using existing model-checking techniques. Second, even when the transformed program remains infinite-state, reducing its randomness can lower output variance and improve the precision of Monte Carlo estimates under a fixed wall-clock budget (Section 7).

2 Language and Determinization

This section introduces the probabilistic language on which our determinization transformation operates. We present its syntax and operational semantics, define determinization as a source-to-source transformation over annotated expressions, and formalize the distribution of successfully returned values through an output measure.

Syntax. Figure 2 presents the syntax of our language, a call-by-value functional language with recursion, pairs, sums, lists, and probability distributions. The expression reject represents a failed execution path used to encode conditioning on observations. Each distribution expression carries a *distribution mode* G, E, or M that guides determinization, as illustrated by the examples in Section 1. The modes G and E mark sampling sites to preserve or replace by mean evaluations, respectively. Determinization changes E to M, which evaluates the distribution expression to its mean. We present these modes explicitly in the syntax to define the transformation precisely, but users may omit them: the inference procedure in Section 3 assigns these modes subject to the typing rules, which check that the replacements are justified by how sampled values are subsequently used.

$$
\begin{aligned}
e ::= {} & x \mid r \ (r \in \mathbb{R}) \mid () \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{reject} \\
& \mid \lambda x.e \mid \mathsf{rec}\ f\ x.\, e \mid e\, e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \\
& \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid -e \mid e + e \mid e \cdot e \mid e/e \mid e < e \\
& \mid \langle e, e \rangle \mid \mathsf{fst}\, e \mid \mathsf{snd}\, e \\
& \mid \mathsf{inl}\, e \mid \mathsf{inr}\, e \mid \mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}\, x \Rightarrow e_1 \mid \mathsf{inr}\, y \Rightarrow e_2 \\
& \mid [] \mid e :: e \mid \mathsf{case}\ e\ \mathsf{of}\ [] \Rightarrow e_0 \mid x :: xs \Rightarrow e_1 \\
& \mid \mathsf{uniform}^{\alpha}(e, e) \mid \mathsf{gaussian}^{\alpha}(e, e) \\
& \mid \mathsf{poisson}^{\alpha}(e) \mid \mathsf{exponential}^{\alpha}(e) \\
& \mid \mathsf{bernoulli}^{\alpha}(e) \mid \mathsf{discrete}^{\alpha}(e) \\
& \mid \mathsf{beta}^{\alpha}(e, e) \mid \mathsf{gamma}^{\alpha}(e, e). \\
\alpha ::= {} & \mathsf{G} \mid \mathsf{E} \mid \mathsf{M}
\end{aligned}
$$

Fig. 2. Language grammar [Lean]. G and E denote sampling; M denotes mean evaluation.

Operational semantics. Figure 3 gives the call-by-value operational semantics. Values and evaluation contexts are shown in Fig. 3(a). In particular, a distribution's arguments are evaluated to values before it is sampled or its mean is computed.

The deterministic reduction rules are shown in Fig. 3(b). We omit the routine arithmetic and comparison rules here. We model the expression reject as divergence through the infinite reduction sequence reject $\rightsquigarrow$ reject, and thus our semantics does not distinguish rejection from nontermination. Distribution expressions in mode M, introduced by determinization, also reduce deterministically to their means: $D^{\mathsf{M}}(\bar{v}) \rightsquigarrow \mathrm{Mean}_D(\bar{v})$ when $\mathrm{Domain}_D(\bar{v})$, where the primitive distribution table in Fig. 3(c) specifies Domain_D and Mean_D for each supported distribution. A distribution expression whose arguments lie outside its domain becomes stuck, as does division by zero: execution cannot take another step or return a value. In $\mathsf{gaussian}(u, v)$, v denotes the

variance; the parameters λ of the exponential and gamma distributions are rates. The distribution discrete$([p_i]_{i<n})$ returns i with probability p_i for $0 \le i < n$, and returns n with the remaining probability $1 - \sum_{i<n} p_i$.

In contrast to mode M, distribution expressions with modes G and E both sample from $D(\bar{v})$ when the arguments satisfy the domain condition $\text{Domain}_D(\bar{v})$. Since sampling is stochastic, neither mode has a deterministic reduction rule. Their identical sampling behavior is instead specified by the output measure rules in Fig. 3(d), which we describe later in this section.

Determinization. We now define the determinization transformation, with the full inductive definition deferred to Section C.

Definition 2.1 (Determinization ∃∀ℕ). Let $\mathcal{A} = \{\mathsf{G}, \mathsf{E}, \mathsf{M}\}$ be the set of distribution modes. The determinization transformation Determinize$(\cdot)$ is defined structurally on expressions, changing each E-mode distribution to M-mode and leaving all other constructs unchanged. In particular, for every distribution D, mode $\alpha \in \mathcal{A}$, and argument expressions $\bar{e} = (e_1, \ldots, e_n)$,

$$\text{Determinize}(D^{\alpha}(\bar{e})) = \begin{cases} D^{\mathsf{M}}(\text{Determinize}(\bar{e})) & \text{if } \alpha = \mathsf{E}, \\ D^{\alpha}(\text{Determinize}(\bar{e})) & \text{if } \alpha \in \{\mathsf{G}, \mathsf{M}\}, \end{cases}$$

where

$$\text{Determinize}(\bar{e}) = (\text{Determinize}(e_1), \ldots, \text{Determinize}(e_n)).$$

For every non-distribution constructor C,

$$\text{Determinize}(C(e_1, \ldots, e_n)) = C(\text{Determinize}(e_1), \ldots, \text{Determinize}(e_n)),$$

and atomic expressions are unchanged.

Most notably, determinization replaces an E-mode distribution with its mean evaluation and recursively transforms the distribution's arguments, preserving their call-by-value evaluation order. When all sampling sites carry distribution mode E, determinization eliminates all randomness, yielding a deterministic program. We refer to this as *full determinization*. When both E- and G-mode distributions occur, determinization eliminates randomness in the former while retaining randomness in the latter. Accordingly, we refer to this as *partial determinization*. In this case, the resulting program may be discrete or remain continuous, depending on the randomness retained. In particular, if all retained sampling sites are discrete, the transformed program is discrete, even if the original program contains continuous distributions. Conversely, retaining continuous sampling sites may leave the transformed program continuous. Section D gives several simple examples to illustrate full and partial determinization in action.

We note that determinization itself is syntactic: it performs the required replacements, but the type system is what establishes that those replacements are semantically justified. For example, the type system prevents the uniform distribution in

$$\text{let } x = \text{uniform}^{\mathsf{E}}(0, 1) \text{ in } x * x$$

from being assigned mode E, since for X uniform on $[0, 1]$, the original program returns X^2, with expectation $\mathbb{E}[X^2] = \frac{1}{3}$, but replacing X by its mean instead returns $\mathbb{E}[X]^2 = \frac{1}{4}$, changing the expected returned value. We formalize such constraints in Section 3.

Output measures. To characterize the probabilistic behavior of a program, we define a subprobability measure over *returned real values*, as shown in Fig. 3(d). Here, δ_r denotes the Dirac measure at r, defined by $\delta_r(B) = 1$ if $r \in B$ and 0 otherwise. We equip $\mathbb{R}$ with its Borel σ-algebra and let $\mathcal{D}(\mathbb{R})$ denote the set of subprobability measures on this measurable space. Let $Expr$ be the set of all expressions. For each $n \in \mathbb{N}$, the function $\mu^{(n)}_{(\cdot)} : Expr \to \mathcal{D}(\mathbb{R})$ assigns each expression e

(a) Values and evaluation contexts

$$
\begin{aligned}
v &::= r \mid () \mid \mathsf{true} \mid \mathsf{false} \mid \lambda x.e \mid \mathsf{rec}\ f\ x.\ e \mid \langle v, v\rangle \mid \mathsf{inl}\, v \mid \mathsf{inr}\, v \mid [] \mid v :: v \\
C &::= [\,] \mid C\ e \mid v\ C \mid \mathsf{let}\ x = C\ \mathsf{in}\ e \mid \mathsf{if}\ C\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \\
&\quad \mid \langle C, e\rangle \mid \langle v, C\rangle \mid \mathsf{fst}\, C \mid \mathsf{snd}\, C \mid \mathsf{inl}\, C \mid \mathsf{inr}\, C \\
&\quad \mid C :: e \mid v :: C \mid \mathsf{case}\ C\ \mathsf{of}\ \mathsf{inl}\, x \Rightarrow e_1 \mid \mathsf{inr}\, y \Rightarrow e_2 \mid \mathsf{case}\ C\ \mathsf{of}\ [] \Rightarrow e_0 \mid x :: xs \Rightarrow e_1 \\
&\quad \mid -C \mid C \circ e \mid v \circ C \mid \mathrm{D}^{\alpha}(\bar{v}, C, \bar{e}), \qquad \circ \in \{+, \cdot, /, <\}, \quad \alpha \in \{\mathrm{G}, \mathrm{E}, \mathrm{M}\}.
\end{aligned}
$$

(b) Deterministic reduction

$$
\begin{aligned}
(\lambda x.e)\, v &\rightsquigarrow e[x \mapsto v] \\
(\mathsf{rec}\ f\ x.\ e)\, v &\rightsquigarrow e[x \mapsto v, f \mapsto \mathsf{rec}\ f\ x.\ e] \\
\mathsf{let}\ x = v\ \mathsf{in}\ e &\rightsquigarrow e[x \mapsto v] \\
\mathsf{reject} &\rightsquigarrow \mathsf{reject} \\
\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 &\rightsquigarrow e_1 \\
\mathsf{if}\ \mathsf{false}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 &\rightsquigarrow e_2 \\
\mathsf{fst}\, \langle v_1, v_2\rangle &\rightsquigarrow v_1 \\
\mathsf{snd}\, \langle v_1, v_2\rangle &\rightsquigarrow v_2 \\
\mathsf{case}\ \mathsf{inl}\, v\ \mathsf{of}\ \mathsf{inl}\, x \Rightarrow e_1 \mid \mathsf{inr}\, y \Rightarrow e_2 &\rightsquigarrow e_1[x \mapsto v] \\
\mathsf{case}\ \mathsf{inr}\, v\ \mathsf{of}\ \mathsf{inl}\, x \Rightarrow e_1 \mid \mathsf{inr}\, y \Rightarrow e_2 &\rightsquigarrow e_2[y \mapsto v] \\
\mathsf{case}\ []\ \mathsf{of}\ [] \Rightarrow e_0 \mid x :: xs \Rightarrow e_1 &\rightsquigarrow e_0 \\
\mathsf{case}\ v :: w\ \mathsf{of}\ [] \Rightarrow e_0 \mid x :: xs \Rightarrow e_1 &\rightsquigarrow e_1[x \mapsto v, xs \mapsto w] \\
\mathrm{D}^{\mathrm{M}}(\bar{v}) &\rightsquigarrow \mathrm{Mean}_D(\bar{v}) \quad (\mathrm{Domain}_D(\bar{v})).
\end{aligned}
$$

(c) Primitive distributions

Distribution $D(\bar{v})$	$\mathrm{Domain}_D(\bar{v})$	$\mathrm{Mean}_D(\bar{v})$
$\mathsf{uniform}(a, b)$	$a \le b$	$(a+b)/2$
$\mathsf{gaussian}(u, v)$	$v \ge 0$	u
$\mathsf{poisson}(\lambda)$	$\lambda \ge 0$	λ
$\mathsf{exponential}(\lambda)$	$\lambda > 0$	$1/\lambda$
$\mathsf{bernoulli}(p)$	$0 \le p \le 1$	p
$\mathsf{discrete}([p_i]_{i<n})$	$p_i \ge 0,\ \sum_{i<n} p_i \le 1$	$n + \sum_{i<n}(i-n)p_i$
$\mathsf{beta}(a, b)$	$a > 0,\ b > 0$	$a/(a+b)$
$\mathsf{gamma}(k, \lambda)$	$k > 0,\ \lambda > 0$	k/λ

(d) Output measures

$$
\mu_e^{(0)}(B) = \begin{cases} \delta_r(B) & e = r, \quad r \in \mathbb{R} \quad \text{(Return real numbers)} \\ 0 & \text{otherwise} \quad \text{(No real result yet)} \end{cases}
$$

$$
\mu_e^{(n+1)}(B) = \begin{cases} \mu^{(n)}_{C[e_2]}(B) & e = C[e_1], \quad e_1 \rightsquigarrow e_2 \quad \text{(Deterministic step)} \\ \displaystyle\int_{\mathbb{R}} \mu^{(n)}_{C[r]}(B)\, D(\bar{v})(dr) & e = C[\mathrm{D}^m(\bar{v})],\ \mathrm{Domain}_D(\bar{v}),\ m \in \{\mathrm{G}, \mathrm{E}\} \quad \text{(Average over samples)} \\ 0 & \text{otherwise} \quad \text{(No output at this depth)} \end{cases}
$$

$$
\mu_e(B) = \sum_{n=0}^{\infty} \mu_e^{(n)}(B).
$$

Fig. 3. Operational semantics. In (b), we omit the usual arithmetic and comparison reductions; division by zero gets stuck. In (c), the degenerate distributions $\mathsf{uniform}(a, a)$ and $\mathsf{gaussian}(u, 0)$ are the point masses δ_a and δ_u. In (d), the output measures are subprobability measures, and are defined for all expressions, but record only executions that return real values; executions returning values of other types contribute no mass.

its output measure after exactly n reduction steps. Summing over all finite n defines the function $\mu_{(\cdot)} : Expr \to \mathcal{D}(\mathbb{R})$, which assigns each expression its complete output measure. For each Borel set $B \subseteq \mathbb{R}$, $\mu_e^{(n)}(B)$ and $\mu_e(B)$ are the probabilities that e returns a value in B after exactly n steps and after any finite number of steps, respectively. Since only executions that successfully return a real value contribute mass, $\mu_e(\mathbb{R}) \leq 1$, with the missing mass accounting for rejection, divergence, stuck executions, or returns of non-real values. Thus, $\mu_e(\mathbb{R})$ is precisely the probability that e successfully returns a real value.

Although executions that reject, diverge, or become stuck contribute no mass to the output measure, stuck executions require special consideration. Replacing a random sample by its mean may change whether a subsequent operation is defined. For example, a uniform distribution whose upper bound depends on a random sample may be undefined for some sampled values but become well-defined after determinization, which can alter the probability of successful return (see Section K). So to rule out such behavior, we introduce the following domain safety condition, and henceforth assume that all programs we consider are domain safe.

Definition 2.2 (Domain safety). Define $\mathrm{Safe}_n(e)$ recursively on n. Set $\mathrm{Safe}_0(e) = \top$ for every expression e. For $n \geq 0$, let

$$\mathrm{Safe}_{n+1}(e) = \begin{cases} \top, & e \text{ is a value}, \\ \mathrm{Safe}_n(C[e']), & e = C[e_0],\ e_0 \rightsquigarrow e', \\ \mathrm{Domain}_D(\bar{v}) \ \wedge\ \big(\mathrm{Safe}_n(C[r]) \text{ for } D(\bar{v})\text{-almost every } r\big), & e = C[D^{\alpha}(\bar{v})],\ \alpha \in \{\mathsf{G}, \mathsf{E}\}, \\ \bot, & \text{otherwise.} \end{cases}$$

A closed expression e is *domain-safe* if $\mathrm{Safe}_n(e)$ holds for every $n \in \mathbb{N}$.

Domain safety is undecidable in general due to unbounded recursion. We therefore treat it as a semantic assumption, separate from the conditions enforced by our type system.

3 Type System and Inference

3.1 Typing

Whether replacing a sample by its mean is valid depends on how the sampled value is subsequently used. We therefore introduce a type system that checks whether sampled values are used in ways that justify replacing them by their means. Figure 4 gives the typing rules. The judgment $\Gamma \vdash e : \tau$ assigns expression e type τ under context Γ. Alongside standard types for functions, pairs, sums, and lists, real types carry an annotation $m \in \{\mathsf{G}, \mathsf{E}\}$, distinguishing values that remain unchanged under determinization from those that may change:

- $\mathrm{real}^{\mathsf{G}}$ describes real values that determinization leaves unchanged.
- $\mathrm{real}^{\mathsf{E}}$ describes real values that may change under determinization and whose uses are restricted to affine operations.

The subtyping relation $\mathrm{real}^{\mathsf{G}} <: \mathrm{real}^{\mathsf{E}}$ allows values of type $\mathrm{real}^{\mathsf{G}}$ wherever values of type $\mathrm{real}^{\mathsf{E}}$ are permitted, but not conversely. Subtyping extends to compound types as well.

Typing rules. The key principle is that operations must preserve expectations when values of type $\mathrm{real}^{\mathsf{E}}$ are replaced by their means. The rules enforce this as follows:

- **Addition and negation.** Both permit operands of type $\mathrm{real}^{\mathsf{E}}$ because expectation is linear, even when the operands depend on the same random samples.
- **Multiplication and division.** Multiplication requires a left operand of type $\mathrm{real}^{\mathsf{G}}$, while division requires a denominator of type $\mathrm{real}^{\mathsf{G}}$. Determinization leaves these values unchanged, so multiplication by such a value—and division by it when nonzero—is linear in the other

Type grammar

$$\tau ::= \mathsf{unit} \mid \mathsf{bool} \mid \mathsf{real}^m \mid \tau \to \tau \mid \tau \times \tau \mid \tau + \tau \mid \mathsf{list}(\tau) \qquad m ::= \mathsf{G} \mid \mathsf{E}$$

Real typing

$$\frac{r \in \mathbb{R}}{\Gamma \vdash r : \mathsf{real}^m}\ \textsc{Real} \qquad \frac{}{\Gamma \vdash \mathsf{reject} : \tau}\ \textsc{Reject} \qquad \frac{\Gamma \vdash a : \mathsf{real}^m \qquad \Gamma \vdash b : \mathsf{real}^m \qquad \alpha = \mathsf{M} \vee \alpha = m}{\Gamma \vdash \mathsf{uniform}^\alpha(a, b) : \mathsf{real}^m}\ \textsc{Uniform}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{real}^{\mathsf{G}} \qquad \Gamma \vdash e_2 : \mathsf{real}^{\mathsf{G}}}{\Gamma \vdash e_1 < e_2 : \mathsf{bool}}\ \textsc{Cmp} \qquad \frac{\Gamma \vdash u : \mathsf{real}^m \qquad \Gamma \vdash v : \mathsf{real}^{\mathsf{G}} \qquad \alpha = \mathsf{M} \vee \alpha = m}{\Gamma \vdash \mathsf{gaussian}^\alpha(u, v) : \mathsf{real}^m}\ \textsc{Gaussian}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{real}^m \qquad \Gamma \vdash e_2 : \mathsf{real}^m}{\Gamma \vdash e_1 + e_2 : \mathsf{real}^m}\ \textsc{Add} \qquad \frac{\Gamma \vdash e_1 : \mathsf{real}^{\mathsf{G}} \qquad \Gamma \vdash e_2 : \mathsf{real}^m}{\Gamma \vdash e_1 \cdot e_2 : \mathsf{real}^m}\ \textsc{Mul}$$

Subtyping

$$\frac{}{\tau <: \tau}\ \textsc{Refl} \qquad \frac{\tau <: \sigma \qquad \sigma <: \rho}{\tau <: \rho}\ \textsc{Trans} \qquad \frac{}{\mathsf{real}^{\mathsf{G}} <: \mathsf{real}^{\mathsf{E}}}\ \textsc{Real-Sub} \qquad \frac{\sigma_1 <: \tau_1 \qquad \tau_2 <: \sigma_2}{\tau_1 \to \tau_2 <: \sigma_1 \to \sigma_2}\ \textsc{Arrow}$$

Fig. 4. Type grammar, real typing, and subtyping rules. We show selected rules for arithmetic operations, distributions, and subtyping; additional rules for these constructs appear in Section E. For distribution modes $\alpha \in \{\mathsf{G}, \mathsf{E}\}$, the type annotation m must match α; for $\alpha = \mathsf{M}$, either type annotation is permitted. We omit the standard rules for variables, unit and Boolean literals, functions, recursion, application, let bindings, conditionals, pairs, projections, sum injections and case analysis, and list constructors and case analysis.

operand. For example, let $x = \mathsf{uniform}^{\mathsf{G}}(0, 1)$ and $y = \mathsf{gaussian}^{\mathsf{E}}(x, 1)$. For each sampled value of x, y has mean x, so $x * y$ has mean x^2. Determinization leaves x unchanged and replaces y by x, giving exactly x^2. A limitation of this rule is that it rejects multiplication of two E values even when they are independent; supporting this case would require an additional analysis to establish independence between random variables.

- **Comparisons.** Both operands must have type $\mathsf{real}^{\mathsf{G}}$, ensuring that determinization does not change their outcomes and consequently alter control flow.
- **Distribution parameters.** Parameters may have type $\mathsf{real}^{\mathsf{E}}$ when the distribution's mean depends affinely on them. For example, $\mathsf{uniform}^{\mathsf{E}}(0, y)$ permits an upper bound of type $\mathsf{real}^{\mathsf{E}}$ because its mean is $y/2$. Similarly, Gaussian means, Poisson rates, Bernoulli probabilities, discrete probabilities, and gamma shapes may have type $\mathsf{real}^{\mathsf{E}}$. Exponential rates, beta parameters, and gamma rates must have type $\mathsf{real}^{\mathsf{G}}$ because the corresponding means are not affine in those parameters. This type system also requires Gaussian variances to have type $\mathsf{real}^{\mathsf{G}}$, although the Gaussian mean is independent of the variance.

Finally, the distribution rules impose $\alpha \in \{\mathsf{G}, \mathsf{E}, \mathsf{M}\}$: G-mode sampling produces values of type $\mathsf{real}^{\mathsf{G}}$, E-mode sampling produces values of type $\mathsf{real}^{\mathsf{E}}$, and M-mode mean evaluation may produce either $\mathsf{real}^{\mathsf{G}}$ or $\mathsf{real}^{\mathsf{E}}$. In particular, G-mode distributions require parameters of type $\mathsf{real}^{\mathsf{G}}$, preserving their sampling behavior under determinization.

The transformation preserves the types assigned by these rules.

Lemma 3.1 (Determinization preserves typing). *If* $\Gamma \vdash e : \tau$, *then*

$$\Gamma \vdash \mathsf{Determinize}(e) : \tau.$$

3.2 Type Inference

The typing rules in Fig. 4 check whether the distribution modes and type annotations assigned to a program are valid. The type inference procedure determines these assignments, together with the types of expressions. Our goal is to assign mode E to as many sampling sites as possible, subject to the typing rules.

Constraint generation

$$\frac{r \in \mathbb{R}}{\Gamma \vdash r \Rightarrow \mathsf{real}^a \mid \varnothing}\ \textsc{Real} \qquad \frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \mid C_1 \qquad \Gamma \vdash e_2 \Rightarrow \tau_2 \mid C_2}{\Gamma \vdash e_1 + e_2 \Rightarrow \mathsf{real}^a \mid C_1 \cup C_2 \cup \{\tau_1 <: \mathsf{real}^a,\ \tau_2 <: \mathsf{real}^a\}}\ \textsc{Add}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \mid C_1 \qquad \Gamma \vdash e_2 \Rightarrow \tau_2 \mid C_2}{\Gamma \vdash e_1 \cdot e_2 \Rightarrow \mathsf{real}^a \mid C_1 \cup C_2 \cup \{\tau_1 <: \mathsf{real}^{\mathsf{G}},\ \tau_2 <: \mathsf{real}^a\}}\ \textsc{Mul}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \mid C_1 \qquad \Gamma \vdash e_2 \Rightarrow \tau_2 \mid C_2}{\Gamma \vdash e_1 < e_2 \Rightarrow \mathsf{bool} \mid C_1 \cup C_2 \cup \{\tau_1 <: \mathsf{real}^{\mathsf{G}},\ \tau_2 <: \mathsf{real}^{\mathsf{G}}\}}\ \textsc{Cmp}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \mid C_1 \qquad \Gamma \vdash e_2 \Rightarrow \tau_2 \mid C_2}{\Gamma \vdash \mathsf{uniform}(e_1, e_2) \Rightarrow \mathsf{real}^a \mid C_1 \cup C_2 \cup \{\tau_1 <: \mathsf{real}^a,\ \tau_2 <: \mathsf{real}^a\}}\ \textsc{Uniform}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \mid C_1 \qquad \Gamma \vdash e_2 \Rightarrow \tau_2 \mid C_2}{\Gamma \vdash \mathsf{gaussian}(e_1, e_2) \Rightarrow \mathsf{real}^a \mid C_1 \cup C_2 \cup \{\tau_1 <: \mathsf{real}^a,\ \tau_2 <: \mathsf{real}^{\mathsf{G}}\}}\ \textsc{Gaussian} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x \Rightarrow \tau \mid \varnothing}\ \textsc{Var}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \mid C_1 \qquad \Gamma, x : \tau_1 \vdash e_2 \Rightarrow \tau_2 \mid C_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \Rightarrow \tau_2 \mid C_1 \cup C_2}\ \textsc{Let}$$

$$\frac{\Gamma \vdash e_0 \Rightarrow \tau_0 \mid C_0 \qquad \Gamma \vdash e_1 \Rightarrow \tau_1 \mid C_1 \qquad \Gamma \vdash e_2 \Rightarrow \tau_2 \mid C_2}{\Gamma \vdash \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \Rightarrow X \mid C_0 \cup C_1 \cup C_2 \cup \{\tau_0 <: \mathsf{bool},\ \tau_1 <: X,\ \tau_2 <: X\}}\ \textsc{If}$$

Fig. 5. Selected constraint generation rules. The judgment $\Gamma \vdash e \Rightarrow \tau \mid C$ infers type τ and generates constraints C. Each annotation a and type variable X introduced in an individual rule is fresh. At a sampling site, a supplies the inferred annotation; an explicit annotation $m \in \{\mathsf{G}, \mathsf{E}\}$ additionally generates $a = m$. The full rules are given in Section F.

3.2.1 Constraint Generation. We generate constraints using judgments of the form $\Gamma \vdash e \Rightarrow \tau \mid C$, meaning that under context Γ, expression e has type τ provided that the set C of constraints is satisfied. Selected rules appear in Fig. 5; the complete rules are given in Section F. Each rule in the figure recursively infers types for its subexpressions. We use variables $a, b, \ldots$, ranging over $\{\mathsf{G}, \mathsf{E}\}$, to represent unknown annotations on real types, and type variables $X, Y, \ldots$ to represent unknown types. In particular, real^a denotes a real type whose annotation is yet to be determined. We order annotations by $a \leq b$ iff $a = b$ or $(a, b) = (\mathsf{G}, \mathsf{E})$. Thus, $\mathsf{real}^a <: \mathsf{real}^b$ holds exactly when $a \leq b$: the permitted assignments are (G, G), (G, E), and (E, E), while (E, G) is forbidden. Section G walks through an example of how the constraint generation process works more concretely.

3.2.2 Constraint Solving. We solve the generated constraints following Traytel et al. [51], in the phases shown in Fig. 6. A type's *shape* is obtained by erasing its real annotations. Since subtyping relates only types of the same shape, we first unify the shapes on both sides of every subtype constraint. Unification rejects constructor clashes and uses an occurs check to reject cyclic equations such as $X = X \to Y$; either failure causes inference to fail.

(a) Shape unification

Erase real annotations and unify the shapes on both sides of every subtype constraint, rejecting constructor clashes and occurs-check violations. Let θ be the resulting most general shape substitution.

(b) Decoration

Replace each type variable X by θX with a fresh annotation variable at each real leaf. Reuse the same decoration at every occurrence of X; use distinct annotation variables for distinct type variables. Preserve existing real annotations.

(c) Subtype constraint reduction

$$\mathsf{real}^a <: \mathsf{real}^b \;\rightsquigarrow\; \{a \le b\} \qquad \mathsf{list}(\tau) <: \mathsf{list}(\sigma) \;\rightsquigarrow\; \{\tau <: \sigma\}$$

$$\chi <: \chi \;\rightsquigarrow\; \emptyset \quad (\chi \text{ is unit, bool, or a shape variable})$$

$$(\tau_1 \to \tau_2) <: (\sigma_1 \to \sigma_2) \;\rightsquigarrow\; \{\sigma_1 <: \tau_1,\ \tau_2 <: \sigma_2\}$$

$$(\tau_1 \star \tau_2) <: (\sigma_1 \star \sigma_2) \;\rightsquigarrow\; \{\tau_1 <: \sigma_1,\ \tau_2 <: \sigma_2\} \quad (\star \in \{\times, +\}).$$

(d) Mode constraint solving

$$\frac{a \le b \qquad a = \mathsf{E}}{b := \mathsf{E}} \qquad\qquad \frac{a \le b \qquad b = \mathsf{G}}{a := \mathsf{G}}$$

Fig. 6. Constraint solving, in the order (a)–(d). Conflicting mode assignments cause inference to fail. Unassigned annotation variables default to E; remaining shape variables become unit at read-back.

Successful unification yields a most general shape substitution θ. We then replace each type variable X by θX, decorated with a fresh annotation variable at each real leaf. All occurrences of X share this decoration; distinct type variables receive distinct annotation variables even when their shapes agree. Existing real annotations are preserved. Subtype constraint reduction then decomposes the decorated constraints into inequalities $a \le b$, where $\mathsf{G} \le \mathsf{E}$.

Finally, we solve the inequalities by propagating forced assignments: $a \le b$ forces $b = \mathsf{E}$ if $a = \mathsf{E}$, and forces $a = \mathsf{G}$ if $b = \mathsf{G}$. We repeat until no further assignments are forced, rejecting conflicting assignments. Remaining annotation variables are assigned E, maximizing the number of sampling sites that can be determinized. Shape variables left unconstrained by unification are assigned unit when reading back the inferred type. Section H gives an example.

To state the guarantee, a *completion* of an expression e is an expression e' obtained by annotating every unannotated sampling site with G or E, while preserving existing annotations, such that $\vdash e' : \tau$ for some type τ. For two completions, write $e' \le e''$ if they differ only at sampling sites annotated G in e' and E in e''. Thus, larger completions permit at least as much determinization.

Theorem 3.2 (Inference correctness). *If e has a completion, inference returns a completion $\hat{e}$ of e and a type τ with $\vdash \hat{e} : \tau$, and $e' \le \hat{e}$ for every completion e' of e. Otherwise, inference fails.*

4 Soundness

We will now prove our determinization technique sound in the following natural sense: First, it *preserves the expected return whenever that expectation is well-defined*. Second, *determinization never increases the variance of returned values*. We proceed by first stating these results in terms of the output measure μ_e, and then develop a more fine-grained soundness characterization that also applies when the program's expected return is undefined. We state the main soundness theorems throughout this section, leaving the proof strategy to Section 5.

4.1 Global Soundness

A first prerequisite is to agree on the following notions of expectations:

Definition 4.1 (Expectation LEM*).* Let X be a real-valued random variable with probability distribution μ, and write $r^+ = \max(r, 0)$, and $r^- = \max(-r, 0)$. We define

$$\mathbb{E}[X^+] := \int_{\mathbb{R}} r^+ \, \mu(dr) \in [0, \infty] \qquad \mathbb{E}[X^-] := \int_{\mathbb{R}} r^- \, \mu(dr) \in [0, \infty] \qquad \mathbb{E}[X] := \mathbb{E}[X^+] - \mathbb{E}[X^-] \, .$$

The expectation $\mathbb{E}[X]$ is *well-defined in the extended reals* if the two quantities $\mathbb{E}[X^+]$ and $\mathbb{E}[X^-]$ are not both infinite. Moreover, $\mathbb{E}[X]$ is *finite* iff $\int_{\mathbb{R}} |r| \, \mu(dr) < \infty$.

Conditioning on successful return. Recall that μ_e is a *sub*probability measure since diverging or rejected executions contribute no returned value. Thus, let $q_e := \mu_e(\mathbb{R})$ LEM denote the *probability that e successfully returns*. When $q_e > 0$, we consider expectations and variances conditioned on successful return. More precisely, we define

$$\mathbb{E}_{\text{ret}}[e] := \frac{1}{q_e} \left(\int_{\mathbb{R}} r^+ \, \mu_e(dr) - \int_{\mathbb{R}} r^- \, \mu_e(dr) \right) \quad \text{LEM}, \text{ and}$$

$$\text{Var}_{\text{ret}}[e] := \frac{1}{q_e} \int_{\mathbb{R}} \left(r - \mathbb{E}_{\text{ret}}[e] \right)^2 \mu_e(dr) \quad \text{LEM} \, .$$

With these notions at hand, we state our central soundness theorem: Determinization preserves the expected return whenever the original program's global expectation is well-defined.

Theorem 4.2 (Expectation preservation LEM). *Suppose* $\vdash e : \mathsf{real}^{\mathsf{E}}$ *and* e *is domain-safe. If* $q_e > 0$ *and* $\mathbb{E}_{\text{ret}}[e]$ *is well-defined in the extended reals, then: (1)* $q_{\text{Determinize}(e)} > 0$*, (2)* $\mathbb{E}_{\text{ret}}[\text{Determinize}(e)]$ *is well-defined, and (3)* $\mathbb{E}_{\text{ret}}[\text{Determinize}(e)] = \mathbb{E}_{\text{ret}}[e]$.

This theorem requires the original program's expectation to be well-defined. The distinction is important even for simple programs, as the following example illustrates.

Example 4.3 (Undefined Global Expectation). Consider a program that flips a fair coin until tails and, after n preceding heads, returns a sample from $\mathsf{uniform}^{\mathsf{E}}(-2^n, 2^n)$:

```
let sample = rec f bound =>
  if bernoulli[G](1/2) < 1 then uniform[E](-bound, bound)
  else f (2 * bound)
in sample 1
```

For any fixed terminating sequence of retained G-samples, the remaining E-sample has mean zero, so the determinized program returns zero. Globally, however, let N be the number of heads and X the returned value. We have $\Pr(N = n) = 2^{-(n+1)}$. Conditioned on $N = n$, $X \sim \mathsf{uniform}(-2^n, 2^n)$ and hence $\mathbb{E}[|X| \mid N = n] = 2^{n-1}$.[1] Therefore,

$$\mathbb{E}[|X|] \;=\; \sum_{n=0}^{\infty} \Pr(N = n) \mathbb{E}[|X| \mid N = n] \;=\; \sum_{n=0}^{\infty} 2^{-(n+1)} 2^{n-1} \;=\; \sum_{n=0}^{\infty} \frac{1}{4} \;=\; \infty \, .$$

Because the distribution of X conditioned on $N = n$ is symmetric around zero for every n and $\mathbb{E}[|X|] = \infty$, we have $\mathbb{E}[X^+] = \mathbb{E}[X^-] = \infty$. Thus, $\mathbb{E}[X]$ is undefined. Nevertheless, the expected return after fixing any terminating sequence of retained samples is finite. This observation motivates a more fine-grained tracewise soundness guarantee that does not require the global expectation to exist, which we present in the next subsection.

Our second theorem states that determinization never increases the variance of returned values:

[1] First absolute moment. For $U \sim \mathsf{uniform}(-a, a)$, $\mathbb{E}[|U|] = (2a)^{-1} \int_{-a}^{a} |u| \, du = a/2$.

G **traces** $\tau \in \mathrm{Trace}$

$$\tau ::= [\,] \mid (D, r) :: \tau, \qquad p_{D,r}(\tau, s) = ((D, r) :: \tau, s).$$

Refined output measures

$$J_e^{(0)} = \begin{cases} \delta_{([\,],r)} & e = r,\ r \in \mathbb{R}, \\ 0 & \text{otherwise}, \end{cases} \qquad J_e^{(n+1)} = \begin{cases} J^{(n)}_{C[e_2]} & e = C[e_1],\ e_1 \rightsquigarrow e_2, \\ \displaystyle\int_{\mathbb{R}} J^{(n)}_{C[r]}\, D(\bar{v})(dr) & e = C[\mathrm{D}^{\mathsf{E}}(\bar{v})],\ \mathrm{Domain}_D(\bar{v}), \\ \displaystyle\int_{\mathbb{R}} (p_{D,r})_* J^{(n)}_{C[r]}\, D(\bar{v})(dr) & e = C[\mathrm{D}^{\mathsf{G}}(\bar{v})],\ \mathrm{Domain}_D(\bar{v}), \\ 0 & \text{otherwise}. \end{cases}$$

$$J_e = \sum_{n=0}^{\infty} J_e^{(n)}, \qquad \eta_e(A) = J_e(A \times \mathbb{R}), \qquad \mu_e(B) = J_e(\mathrm{Trace} \times B).$$

Fig. 7. Trace refinement of Fig. 3. $J_e^{(n)}$ records the trace and output of executions that successfully return after exactly n steps. An empty trace is returned at a real value, an E-sample is integrated out without changing the trace, while $(p_{D,r})_*$ prepends (D, r) to each trace. Sampling requires $\mathrm{Domain}_D(\bar{v})$. Summing over n gives J_e, whose trace and output marginals are η_e and μ_e, respectively.

Theorem 4.4 (Variance non-increase). *Suppose* $\vdash e : \mathsf{real}^{\mathsf{E}}$ *and* e *is domain-safe. If* $q_e > 0$ *and* $\int_{\mathbb{R}} r^2\, \mu_e(dr) < \infty$, *then: (1)* $q_{\mathrm{Determinize}(e)} > 0$, *(2)* $\mathrm{Determinize}(e)$ *has a finite second moment, and (3)* $\mathrm{Var}_{\mathrm{ret}}[\mathrm{Determinize}(e)] \le \mathrm{Var}_{\mathrm{ret}}[e]$.

4.2 Tracewise Soundness

Although determinization is a syntactic transformation, its semantic effect can be characterized by conditioning on the random choices that it retains, namely, the samples annotated with G. Intuitively, once the outcomes of these G-mode samples are fixed, the remaining randomness comes only from the E-mode samples that determinization replaces by their means. Under these fixed G choices, the transformed program returns the expected output of the original program over the remaining E-mode randomness.

This conditional characterization is the key to our soundness result and is useful for two reasons. First, it gives a local account of what determinization does even when the program's overall expectation is undefined. Second, when the global expectation does exist, expectation preservation follows by integrating over these fixed choices. Throughout, a property $P(\tau)$ holds μ-almost everywhere if $\{\tau \mid \neg P(\tau)\}$ is contained in a measurable set of μ-measure zero.

We begin by making precise what it means to fix the random choices retained by determinization.

Definition 4.5 (Trace). Let Prim denote the set of distribution primitives listed in Fig. 3(c). We define the trace space $\mathrm{Trace} := \bigsqcup_{n \in \mathbb{N}} (\mathrm{Prim} \times \mathbb{R})^n$, equipped with the disjoint-union Borel σ-algebra. A *trace* $\tau \in \mathrm{Trace}$ is a finite sequence

$$\tau ::= [\,] \mid (D, r) :: \tau,$$

where each pair (D, r) records the distribution primitive D and sampled value r of an executed G-mode sampling operation, in execution order.

Note that E-mode samples do not appear in a trace: they remain random after the trace has been fixed. The recorded G-mode samples in a trace may come from discrete or continuous distributions. The probability of a trace is the probability that the program successfully terminates with exactly that sequence of sampling outcomes. We remark that this probability can be zero, as when the trace specifies the exact outcome of a sample from a continuous distribution.

Now, we refine the output semantics to jointly record the trace and the returned value, as shown in Fig. 7. Like the original output measure μ_e, this refined semantics records only executions that successfully return a real value and is therefore a subprobability measure; rejected or diverging executions contribute no output pair. Let $\mathcal{D}(\mathsf{Trace}\times\mathbb{R})$ denote the space of subprobability measures on $\mathsf{Trace}\times\mathbb{R}$. For each $n\in\mathbb{N}$, the function $J^{(n)}_{(\cdot)} : Expr \to \mathcal{D}(\mathsf{Trace}\times\mathbb{R})$ assigns each expression e its joint output measure after *exactly* n reduction steps. E-mode samples are integrated out, whereas each G-mode sample contributes its distribution primitive D and sampled value r to the trace τ. Summing these measures over all $n\in\mathbb{N}$ gives rise to the function $J_{(\cdot)} : Expr \to \mathcal{D}(\mathsf{Trace}\times\mathbb{R})$, which assigns each expression e its joint big-step subprobability measure J_e ∃∀. Intuitively, $J_e(A\times B)$ is the probability mass of successfully returning executions whose trace lies in A and whose returned value lies in B.

The two marginals of J_e recover our quantities of interest. For each measurable $A\subseteq\mathsf{Trace}$, its trace marginal $\eta_e(A)$ ∃∀ is the probability mass of successful executions whose trace lies in A. In particular, $\eta_e(\{\tau\})$ is the probability that e successfully terminates with exactly the trace τ. Similarly, for each Borel set $B\subseteq\mathbb{R}$, its output marginal $\mu_e(B)$ ∃∀ is the probability mass of successful executions whose returned value lies in B. This is precisely the output measure defined in Fig. 3(d). Evaluating both marginals on their full spaces gives the following:

Theorem 4.6 (Equality of total masses ∃∀). *For every expression e,*

$$\eta_e(\mathsf{Trace}) = J_e(\mathsf{Trace}\times\mathbb{R}) = \mu_e(\mathbb{R}).$$

Thus both marginals have the same total mass: the probability that e successfully returns.

Let us first consider how determinization affects the trace marginal η_e. Determinization preserves both domain safety and this trace marginal:

Lemma 4.7 (Trace preservation ∃∀). *Suppose $\vdash e : \mathsf{real}^{\mathsf{E}}$ and e is domain-safe. Then:*

(1) $\mathrm{Determinize}(e)$ *is domain-safe, and*

(2) $\eta_{\mathrm{Determinize}(e)} = \eta_e$.

Conditioning on traces. The joint semantics not only recovers the trace and output marginals. Conditioning the joint big-step measure J_e on its trace component gives the output distribution after a trace has been fixed. Since Trace and $\mathbb{R}$ are standard Borel spaces, the disintegration theorem guarantees the existence of a conditional output distribution, unique η_e-almost everywhere. For measurable $A\subseteq\mathsf{Trace}$ and $B\subseteq\mathbb{R}$, we denote this distribution by $\mu_e(\cdot\mid\tau)$, satisfying

$$J_e(A\times B) = \int_A \mu_e(B\mid\tau)\,\eta_e(d\tau)\ . \tag{1}$$

Unlike conditioning by dividing by the probability of an individual trace, this characterization remains valid even when individual traces have probability zero. On a set of traces of η_e-measure zero, the conditional distribution is not uniquely determined, but its value does not affect the integral above. We write

$$\mathbb{E}[e\mid\tau] := \int_{\mathbb{R}} r^{+}\,\mu_e(dr\mid\tau) - \int_{\mathbb{R}} r^{-}\,\mu_e(dr\mid\tau)$$

for the expectation of $\mu_e(\,\cdot\mid\tau)$, which is the expected return value of the original program after fixing a trace τ and averaging over the remaining E-mode randomness. We can now state tracewise soundness.

Theorem 4.8 (Tracewise soundness ∃∀∃). *Suppose* $\vdash e : \mathsf{real}^{\mathsf{E}}$ *and* e *is domain-safe. For* η_e*-almost every trace* τ*,*

$$\int_{\mathbb{R}} |r|\, \mu_e(dr \mid \tau) < \infty \qquad \text{and} \qquad \mu_{\mathrm{Determinize}(e)}(\,\cdot \mid \tau) = \delta_{\mathbb{E}[e|\tau]}\,.$$

Moreover, for η_e*-almost every trace* τ*,*

$$\mathbb{E}[\mathrm{Determinize}(e) \mid \tau] = \mathbb{E}[e \mid \tau]\,.$$

This theorem statement does not require the program to have a well-defined global expectation. For a concrete example, Section I illustrates how tracewise soundness applies. Integrating tracewise soundness (Theorem 4.8) over the common trace measure yields expectation preservation (Theorem 4.2) when the global expectation is well-defined, and variance non-increase (Theorem 4.4) when the second moment is finite. More precisely, if $q_e > 0$ and $\mathbb{E}_{\mathrm{ret}}[e]$ is well-defined, the disintegration identity (Eq. (1)) gives

$$\mathbb{E}_{\mathrm{ret}}[e] = \frac{1}{q_e} \int_{\mathrm{Trace}} \mathbb{E}[e \mid \tau]\, \eta_e(d\tau).$$ ∃∀∃

Note that such a normalization factor was not needed for the conditional output distribution from Theorem 4.8, since η_e records only traces of successfully returning executions, so $\mu_e(\mathbb{R} \mid \tau) = 1$ for η_e-almost every trace τ. Trace preservation immediately implies that determinization preserves the probability of successful return, meaning that the original and transformed programs are conditioned on the same returning mass:

$$q_{\mathrm{Determinize}(e)} = \eta_{\mathrm{Determinize}(e)}(\mathsf{Trace}) = \eta_e(\mathsf{Trace}) = q_e.$$ ∃∀∃

Finally, we investigate how determinization affects the probabilities of successful return, rejection, and divergence. The results above concern the distribution of successfully returned values, but determinization also preserves the probability of returning at all. Let d_e denote the probability of divergence ∃∀∃, including rejection. Under domain safety, executions cannot lose probability mass by getting stuck. Together with preservation of output mass, this implies that determinization also preserves the total probability of divergence and rejection.

Proposition 4.9 (Probabilistic type safety ∃∀∃). *Suppose* $\vdash e : \mathsf{real}^{\mathsf{E}}$ *and* e *is domain-safe. Then* $\mathrm{Determinize}(e)$ *is domain-safe, and*

$$\mu_e(\mathbb{R}) + d_e = 1$$ ∃∀∃, $$\qquad \mu_{\mathrm{Determinize}(e)}(\mathbb{R}) = \mu_e(\mathbb{R})$$ ∃∀∃, $$\qquad d_{\mathrm{Determinize}(e)} = d_e.$$

5 Symbolic Semantics

In this section, we present our proof strategy for establishing tracewise soundness, from which the global soundness results follow. We use the signal example from Section 1 as a running example to illustrate our main ideas, referring to parts in Figure 8. The central idea is to introduce a *symbolic operational semantics* that provides a common representation of both the original and determinized programs. Instead of immediately sampling E-mode distributions, symbolic execution records these choices and postpones their evaluation. Two interpretations of the resulting symbolic states then recover the source program and its determinization.

Symbolic execution. Let us first introduce the notion of a symbolic state.

Definition 5.1 (Symbolic state). A symbolic state s has the form

$$s ::= (\sigma \mid \widehat{e}), \qquad \widehat{e} \in \widehat{\mathrm{Expr}},$$

$$\sigma ::= \emptyset \mid \sigma,\ u \sim D(\bar{a}), \qquad a = c_0 + \sum_{i=1}^{k} c_i u_i, \qquad c_0, \ldots, c_k \in \mathbb{R}.$$

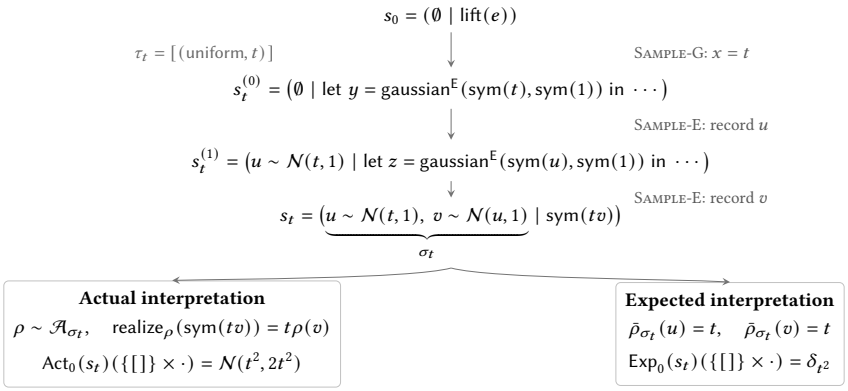


Fig. 8. Symbolic execution of the signal example e from Section 1. The G-mode sample produces $t \sim \mathrm{uniform}(0,1)$ and the trace τ_t, whereas the two E-mode samples are recorded in the symbolic environment. Ordinary reductions (including `let` substitutions) are omitted between states. The terminal state has two interpretations: the source output distribution (left) and its mean (right). Both terminal interpretations record only the empty *remaining* trace [], since the sample t has already been made.

Here, $\widehat{e}$ is a residual symbolic expression ∃∀N, whose real literals may be affine expressions a ∃∀N. The symbolic environment σ ∃∀N is an ordered sequence of declarations $u \sim D(\bar a)$, where u is fresh and the parameters $\bar a$ may refer only to earlier variables in σ.

Consider the running example, which samples x from a uniform distribution and then samples two dependent Gaussian variables y and z before returning xz. Suppose the G-mode sample produces $x = t$. Symbolic execution records the Gaussian samples as fresh variables u and v, eventually reaching the terminal state shown in Fig. 8:

$$s_t = (\sigma_t \mid \mathsf{sym}(tv)), \qquad \sigma_t = \big(u \sim \mathcal{N}(t,1),\ v \sim \mathcal{N}(u,1)\big).$$

The environment σ_t records the dependence of v on u without sampling either variable. The residual expression $\mathsf{sym}(tv)$ is affine in v, since the retained sample t has already been fixed.

Each environment σ induces a joint distribution $\mathcal{A}_\sigma$ ∃∀N over valuations ρ by sampling its declarations in order: for $\sigma = (u_i \sim D_i(\bar a_i))_{i=1}^k$, the distribution $\mathcal{A}_\sigma$ samples $\rho(u_i) \sim D_i(\bar a_i[\rho])$ sequentially. For this example,

$$\mathcal{A}_{\sigma_t}(du, dv) = \mathcal{N}(t,1)(du)\,\mathcal{N}(u,1)(dv).$$

Symbolic execution ∃∀N operates on these symbolic states by reducing residual expressions and extending the symbolic environment with declarations for E-mode distributions. Ordinary reductions operate on symbolic expressions (Reduce), while an E-mode draw adds a declaration to σ without sampling it (Sample-E), as illustrated by the two Gaussian declarations in Fig. 8. A G-mode draw instead samples a concrete value (Sample-G), as illustrated by the uniform draw $x = t$ in the same figure ∃∀N. The reduction rules and their full definitions are given in Section J.4.

5.1 Two Interpretations of the Symbolic Semantics

A symbolic state can be interpreted in two ways. The *actual interpretation* samples the declarations in σ according to $\mathcal{A}_\sigma$, recovering the source program's behavior. The *expected interpretation* instead replaces each symbolic variable by the mean of its declared distribution.

Since distributions may depend on earlier variables, these means are evaluated sequentially. For $\sigma = (u_i \sim D_i(\bar{a}_i))_{i=1}^k$, define the *sequential mean valuation* $\bar{\rho}_\sigma$ by

$$\bar{\rho}_\sigma(u_i) := \mathrm{Mean}_{D_i}(\bar{a}_i[\bar{\rho}_\sigma]), \qquad i = 1, \ldots, k.$$

In our example, this gives $\bar{\rho}_{\sigma_t}(u) = t$ and $\bar{\rho}_{\sigma_t}(v) = t$. Thus, while the actual interpretation samples v according to its dependent Gaussian distribution, the expected interpretation replaces it by t, making the returned value t^2, as shown in the two interpretation boxes in Fig. 8.

To define these interpretations formally, we use two maps. Lifting lift embeds a source expression into a residual expression, replacing each real literal r with $\mathrm{sym}(r)$. Realization $\mathrm{realize}_\rho$ evaluates symbolic literals under a valuation ρ:

$$\mathrm{realize}_\rho(\mathrm{sym}(a)) = a[\rho].$$

Both maps otherwise preserve expression constructors. Symbolic execution begins at $\mathrm{init}(e) := (\emptyset \mid \mathrm{lift}(e))$, and $\mathrm{realize}_{()}(\mathrm{lift}(e)) = e$. The full definitions appear in Section J.6.

Definition 5.2 (Actual and expected interpretations). For $s = (\sigma \mid \widehat{e})$ and $n \in \mathbb{N}$, define

$$\mathrm{Act}_n(s) := \int J^{(n)}_{\mathrm{realize}_\rho(\widehat{e})}\, \mathcal{A}_\sigma(d\rho), \qquad \mathrm{Exp}_n(s) := J^{(n)}_{\mathrm{Determinize}(\mathrm{realize}_{\bar{\rho}_\sigma}(\widehat{e}))}.$$

Both interpretations are measures on $\mathrm{Trace} \times \mathbb{R}$, recording executions that return after exactly n remaining steps. At initialization, they recover the source and determinized programs: $\mathrm{Act}_n(\mathrm{init}(e)) = J^{(n)}_e$ and $\mathrm{Exp}_n(\mathrm{init}(e)) = J^{(n)}_{\mathrm{Determinize}(e)}$. Returning to s_t, the actual interpretation samples u and v, while the expected interpretation substitutes their sequential means. Since v has marginal distribution $\mathcal{N}(t, 2)$, we obtain

$$\mathrm{Act}_0(s_t)(\{[]\} \times \cdot) = \mathcal{N}(t^2, 2t^2), \qquad \mathrm{Exp}_0(s_t)(\{[]\} \times \cdot) = \delta_{t^2}.$$

The two interpretations have different output distributions, but the mean under the actual interpretation equals the value returned by the expected interpretation. This equality follows from the affine-means identity: for a well-typed, domain-safe symbolic environment σ and affine expression a,

$$\int a[\rho]\, \mathcal{A}_\sigma(d\rho) = a[\bar{\rho}_\sigma].$$

In particular, the equality holds even when the distributions in σ depend on earlier samples. Together with the typing restrictions, the affine-means identity relates the actual and expected interpretations throughout symbolic execution.

5.2 Soundness Results

Finite-step factorization. The following theorem relates the two interpretations through a trace-indexed output kernel $K_{n,s}(\tau, \cdot)$. Given a trace τ of G-mode samples, the kernel describes the source output distribution after n remaining steps, integrating over the E-mode samples.

Theorem 5.3 (Finite-step factorization). *Let $s = (\sigma \mid \widehat{e})$ be a well-typed, domain-safe symbolic state with $\widehat{e} : \mathsf{real}^{\mathsf{E}}$. For every $n \in \mathbb{N}$, there exists a kernel $K_{n,s}$ such that, for all measurable $A \subseteq \mathrm{Trace}$, $B \subseteq \mathbb{R}$,*

$$\mathrm{Act}_n(s)(A \times B) = \int_{A \times \mathbb{R}} K_{n,s}(\tau, B)\, \mathrm{Exp}_n(s)(d\tau, dr).$$

Moreover, for $\mathrm{Exp}_n(s)$-almost every (τ, r), $K_{n,s}(\tau, \cdot)$ is a probability measure with finite first absolute moment, and $r = \int_{\mathbb{R}} x\, K_{n,s}(\tau, dx)$.

The first equality recovers the actual interpretation by restoring the randomness removed by the expected interpretation. The mean identity states that the determinized output is the mean of $K_{n,s}(\tau, \cdot)$. For our terminal state, for example,

$$K_{0,s_t}([], \cdot) = \mathcal{N}(t^2, 2t^2), \qquad \int_{\mathbb{R}} r \, K_{0,s_t}([], dr) = t^2.$$

From finite steps to whole programs. The whole-program kernel $K_e(\tau, \cdot)$ ∃∀N describing the source program's output distribution for a trace τ, compatibility lemma ∃∀N, and factorization proof ∃∀N are formalized in Lean. We state the resulting theorem here.

Theorem 5.4 (Whole-program factorization ∃∀N). *Suppose* $\vdash e : \mathsf{real}^{\mathsf{E}}$ *and* e *is domain-safe. There exists a kernel* K_e *such that, for all measurable* $A \subseteq \mathsf{Trace}$ *and* $B \subseteq \mathbb{R}$,

$$J_e(A \times B) = \int_{A \times \mathbb{R}} K_e(\tau, B) \, J_{\mathrm{Determinize}(e)}(d\tau, dr).$$

Moreover, for $J_{\mathrm{Determinize}(e)}$*-almost every* (τ, r), $K_e(\tau, \cdot)$ *is a probability measure with finite first absolute moment, and* $r = \int_{\mathbb{R}} x \, K_e(\tau, dx)$.

Tracewise soundness (Theorem 4.8), including the identification of K_e with the source's conditional output distribution almost everywhere, is proved in Lean ∃∀N.

For the running example, fixing the retained trace $\tau_t = [(\mathsf{uniform}, t)]$ gives

$$\mu_e(\cdot \mid \tau_t) = \mathcal{N}(t^2, 2t^2), \qquad \mu_{\mathrm{Determinize}(e)}(\cdot \mid \tau_t) = \delta_{t^2}.$$

Integrating over $t \sim \mathcal{U}(0, 1)$, both programs have expected output $1/3$, while determinization removes the conditional Gaussian variance. More generally, integrating over the common trace measure gives expectation preservation (Theorem 4.2) whenever the global expectation is well-defined ∃∀N. The law of total variance then gives variance non-increase (Theorem 4.4) when the second moment is finite ∃∀N, establishing the global soundness theorems in Section 4.

6 Implementation

Every theorem of the previous sections is mechanized in the Lean programming language [25], using Mathlib's formalization of measure theory and probability [43], in particular the Giry monad, Markov kernels and their disintegration, as well as the standard continuous and discrete distributions. The complete compilation, verification and execution pipeline is outlined in Fig. 9.

First, the source program is parsed and desugared; this step is unverified. Type inference then fills in every omitted E/G annotation and returns a typed program e_{src}. By Theorem 3.2, it returns the greatest completion, which has an E at every site where some completion has one. The determinization function of Definition 2.1 then replaces every E-mode sample by a mean site, marked M, and returns a well-typed program e_{det}.

The soundness theorems (Theorems 4.2, 4.4 and 4.8) relate e_{src} and e_{det} through the measure-theoretic semantics of Fig. 3, which we define in Lean with the Giry monad. To draw samples, an unverified interpreter runs either program with floating-point numbers and pseudo-random samplers. The theorems do not cover it, and it cannot be proved to agree with the semantics: floating-point numbers are not real numbers, so its results only approximate those of the semantics.

If a program has finitely many reachable states and draws only from Bernoulli and discrete distributions, its output law is that of a finite Markov chain, and its return probability and first and second moments solve linear equations over that chain. The tool builds the chain by exploring the machine states of the program and merging equal ones. Numbers added to the result of a pending call become rewards on transitions, so that loops that accumulate a sum stay finite. A solver verified in Lean, or alternatively Storm [32], solves the equations exactly. From either solver's solution,

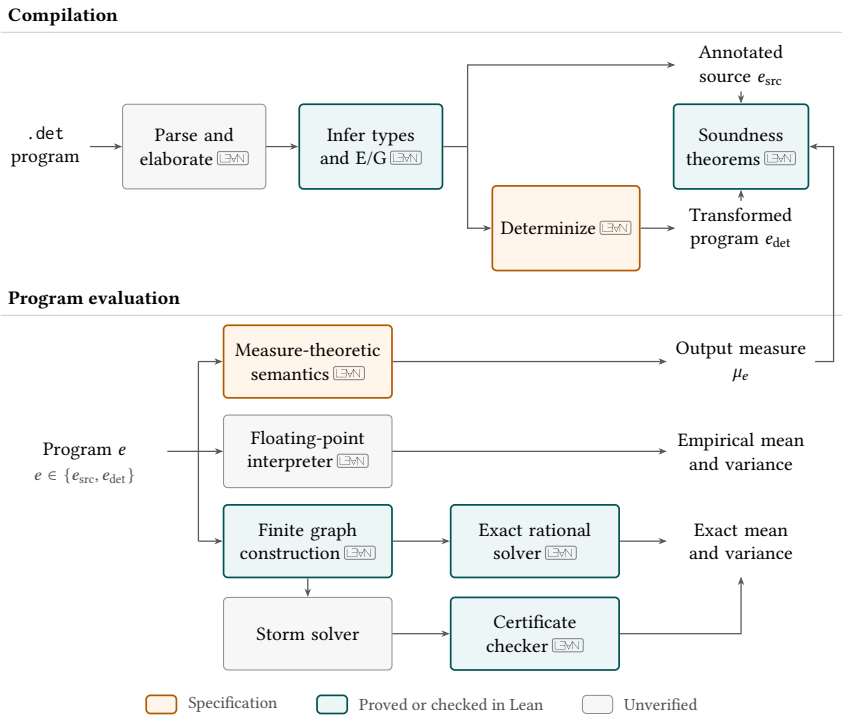


Fig. 9. Implementation architecture. Box styles say how each stage is justified; Lean marks link to the code.

the tool writes a certificate that Lean's kernel checks, which proves that the explored program is domain-safe, that its output law is that of the chain, and that the numbers are its exact return probability and first and second moments. It also proves the chain's probabilities of rejection and of divergence, which add up to one with the return probability. The tool reports these moments, from which follow the probability that the program returns, and the mean and variance of the result given that it returns. If the explored program is e_{det} and e_{src} is domain-safe with a defined expectation, Theorem 4.2 guarantees that e_{src} has the same mean. Chatterjee et al. [21] have Storm produce certificates and check them with a checker verified in Isabelle/HOL. Their certificates cover Markov decision processes, a more general class of models than our Markov chains, but the construction of their models is not verified; here, the chain is checked against the program's semantics, and the certificates also cover the second moment.

Most of the Lean development, including the definitions and theorem statements, was written by large language models (Claude Fable 5.1, Claude Opus 5.5 and GPT 6 Astra). Since Lean's kernel checks the proofs but not whether a statement says what this paper says, we separated the code into two parts:

(1) **Specification**: the definitions in `Spec/`, such as the syntax, typing rules, semantics and determinization, and the full statement of every theorem in `Theorems.lean`, about 1,500 lines in total. We reviewed both line by line against Sections 2 to 4, and reviewed every later change to them.

(2) **Proofs and procedures**: about 27,000 lines in `Proof/` that prove the statements, and the executable code that some statements are about, such as type inference. `Theorems.lean` proves each theorem by a single reference to this part. Its last command fails the build if a

theorem uses an axiom other than `propext`, `Classical.choice` and `Quot.sound`, or if its statement depends, through anything other than proofs, on a declaration in `Proof/`.

To trust the tool's results, one has to rely on Lean's kernel, these three axioms, the specification and the Mathlib definitions it uses, and the Lean compiler for the output of type inference. Exact results come with a certificate that the kernel checks against the specification. The parser and the floating-point interpreter are unverified.

To show interactively how determinization works, we also provide a browser-based simulator, an unverified TypeScript port of the tool's parser, type inference, determinization, interpreter and finite graph construction. Its editor shows the E/G annotations that type inference fills in, next to the determinized program. It steps through a run of the source program, the symbolic state of Section 5 and the determinized program side by side, sharing their G draws. It also runs both programs many times and shows their empirical output distributions, means and variances, next to their exact values where the graph is finite. Its tests compare it with the Lean tool on every program of our test corpus.

7 Evaluation

In this section, we evaluate our determinization approach in three complementary ways. We seek to address the following research questions:

- **RQ1: Exact analysis.** Can determinization transform infinite-state programs into finite-state ones amenable to exact expectation computation, and what is the computational cost of the determinization transformation?
- **RQ2: Variance reduction.** How much does determinization reduce output variance?
- **RQ3: Monte Carlo efficiency.** Does determinization yield more precise Monte Carlo expectation estimates at comparable sampling cost?

Consequently, our first question investigates whether determinization enables computing an exact expectation of a program, through finite-state model checking. The latter two investigate its benefits for sampling-based inference, first measuring reductions in output variance and then comparing Monte Carlo precision and sampling time. We detail the benchmarks, experimental setup, comparisons, and results for each question in the following subsections below. All experiments ran on a MacBook Pro M4 Max with 48GB RAM. Together with our benchmarks, our implementation is publicly available at https://github.com/[anonymized].

7.1 RQ1: Enabling Exact Expectations

We first ask whether determinization can make calculating exact expectations possible for programs with continuous or infinite-state behavior. For the benchmarks in Table 1, determinization yields a finite-state representation, even when some discrete sampling remains. For these benchmarks, we translate the determinized program into a finite discrete-time Markov chain and use Storm [32] to compute its expectation exactly as an expected-reward query. Each result is accompanied by a Lean-checkable certificate establishing that the finite model represents the determinized program and that the reported expectation satisfies the exact rational reward equations and boundary conditions.

Our benchmark suite draws primarily from prior work [6, 11, 17, 18, 28, 50, 52], alongside four new examples: `retransmit`, `service-policy`, `workload`, and `noisy-iteration`. These benchmarks exercise a range of language constructs, including: lists for `add-noise`, `gauss-random-walk`, and `workload`; higher order functions for `monte-carlo` and `service-policy`; distributions with nonconstant parameters for `uniform-nested`; bounded recursion for `aggregate-rv`, `dreckon`, `invpend`, and `swap`; and unbounded recursion for `add-uniform-counter`, `asym-random-walk`,

pack, retransmit, and noisy-iteration. The complete set of benchmark descriptions and programs is Section L.

Benchmark	Det.	Expectation	Sites	Det. time (ms)
add-noise [6]	●●	$\mathbb{E}[sum]$	$(0,1) \to (0,0)$	0.157
add-uniform-counter [11]	●○	$\mathbb{E}[x]$	$(1,2) \to (1,0)$	0.550
aggregate-rv [18]	●●	$\mathbb{E}[x_n]$	$(0,1) \to (0,0)$	0.123
asym-random-walk [11]	●○	$\mathbb{E}[t]$	$(1,3) \to (1,0)$	0.126
dreckon [17]	●○	$\mathbb{E}[x]$	$(1,14) \to (1,0)$	0.799
gauss-random-walk [52]	●●	$\mathbb{E}[y_6]$	$(0,1) \to (0,0)$	0.221
invpend [17]	●●	$\mathbb{E}[pAng_n]$	$(0,6) \to (0,0)$	0.301
monte-carlo [50]	●●	$\mathbb{E}[estimate]$	$(0,1) \to (0,0)$	0.172
noisy-iteration	●○	$\mathbb{E}[x]$	$(1,1) \to (1,0)$	0.054
pack [17]	●○	$\mathbb{E}[weight_n]$	$(1,3) \to (1,0)$	0.697
retransmit	●○	$\mathbb{E}[ret]$	$(4,3) \to (4,0)$	0.098
service-policy	●○	$\mathbb{E}[ret_n]$	$(4,4) \to (4,0)$	0.244
swap [18]	●●	$\mathbb{E}[x_n]$	$(2,2) \to (0,0)$	0.144
uniform-nested [28]	●●	$\mathbb{E}[d]$	$(0,4) \to (0,0)$	0.040
workload	●○	$\mathbb{E}[ret_n]$	$(1,2) \to (1,0)$	0.131

Table 1. Determinization on 15 benchmarks. In the determinization column, ●● denotes full determinization and ●○ denotes partial. Each pair (d, c) counts static discrete and continuous sampling sites. Timings measure determinization only, excluding finite-state model construction and Storm model checking. Several programs are parameterized by n, which affects their expectations and generated model sizes.

Results. Table 1 summarizes our results. For each benchmark, we report the queried expectation, whether determinization is full or partial, the number of static discrete and continuous sampling sites before and after determinization, and the determinization time. All 15 benchmarks have their continuous sampling sites eliminated, with seven fully determinized and eight partially determinized. Even when discrete sampling remains, the resulting programs admit an exact expectation computation using Storm since they are finite-state. These results demonstrate that determinization can allow programs involving continuous distributions, higher-order functions, and unbounded recursion to become finite-state.

Determinization cost. The final column of Table 1 reports the total determinization time. This takes less than one millisecond on every benchmark instance. Now, to investigate how this cost grows with program size, we take three benchmarks from the table and parameterize them by k. For uniform-nested, k counts a chain of dependent uniform sampling sites, each parameterized by the preceding sample. For gauss-random-walk and add-noise, k instead counts the number of explicitly constructed input list elements (see Section M for full details). Figure 11 in Section M plot the median times over 10 runs.

7.2 RQ2: Reducing Output Variance

We next measure how much determinization reduces the variability of individual program outputs. Specifically, we measure the *output variance*: how much the values returned by repeated executions of a program vary around their mean. Recall that for a return value X, the output variance is:

$$\mathrm{Var}(X) = \mathbb{E}\left[(X - \mathbb{E}[X])^2\right].$$

Since $\mathrm{Var}(X)$ is generally not known analytically, we estimate it from repeated independent executions of the program. For K returned values $X_1, \ldots, X_K$, the sampler reports the empirical

variance:

$$\widehat{\mathrm{Var}}(X) = \frac{1}{K}\sum_{i=1}^{K}(X_i - \bar{X})^2, \qquad \bar{X} = \frac{1}{K}\sum_{i=1}^{K} X_i.$$

We apply the same Monte Carlo procedure to the source program P and its determinization $\mathrm{det}(P)$, with 50,000 executions per program. Rejected observations contribute no returned value, so these statistics are conditional on successful return.

Our suite contains 14 benchmarks drawn primarily from prior literature [8, 9, 11, 26, 34, 45], with five new benchmarks we added: `hmm`, `load`, `branching`, `dungeon`, and `noisy-product`. Full programs are given in Section N. We measure how determinization reduces output variance, including for programs that remain infinite-state after the transformation. In some benchmarks, determinization eliminates continuous randomness entirely, while in others it removes only some continuous sampling sites.

Results. Our results are shown in Table 2. In particular, determinization reduces output variance on 12 of the 14 benchmarks and leaves it unchanged on the remaining two. The reduction ranges from modest improvements on `growing-walk` (1.09×) and `2d-walk` (1.32×) to substantially larger gains on `dungeon` (13.02× estimated, versus approximately 12.36× analytically), `load` (40.47×), and `hmm` (108.13×). On `cont-sampling` and `neals-funnel`, determinization removes all variance in the returned value, giving an infinite variance-reduction factor. On `vasicek`, the random stopping time leaves nonzero variance, with a reduction of 1.90×. For `ckd-epi` and `figure1`, the eliminated draws are unused, so eliminating them leaves the output distribution unchanged.

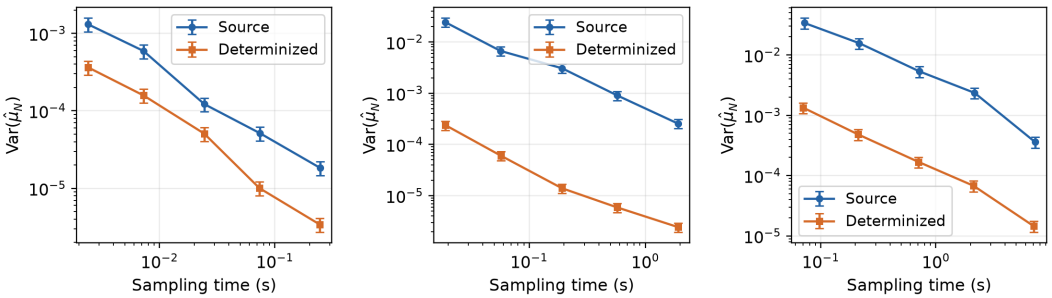
7.3 RQ3: Improving Monte Carlo Efficiency

A reduction in sample complexity does not necessarily translate into the same reduction in wall-clock time, since determinization may also change the cost of generating each sample. For example, a program may have a large variance-reduction factor but more expensive determinized executions, making its practical speedup smaller than its VRF. Conversely, even a moderate reduction in variance may yield a larger practical improvement if samples from the determinized program are cheaper to generate. We therefore next evaluate whether determinization improves the *wall-clock efficiency* of Monte Carlo estimation. We focus on three representative benchmarks—`branching`, `hmm`, and `load`—that exhibit substantial but finite variance reductions in RQ2.

Results. Figure 10 shows lower empirical estimator variance for the determinized programs across all tested sample sizes, with similar sampling times to the source programs. At $N = 10^5$, estimator variance is reduced by approximately 5.4× for `branching`, 105× for `hmm`, and 24.9× for `load`. Thus, on these three benchmarks, the variance reductions translate into more precise Monte Carlo estimates at comparable sampling cost.

8 Related Work

Expectation and Moment Analysis. Several PPLs directly compute or estimate expectations and moments. Hakaru supports calculating exact symbolic expectations via program transformations [57], but does not support higher order constructs. Psi performs exact symbolic inference over discrete and continuous distributions and can answer expectation queries from the resulting exact posterior [28]; λPsi extends this approach with tuples, arrays, higher-order functions, and nested inference [29], but its loops must be statically bounded and are unrolled during inference. Polar computes exact (higher) moments of program variables by deriving and solving recurrence equations for probabilistic loops [45]. Its analysis is nevertheless specialized to loop programs and does not provide general recursion, higher-order functions, or recursive data structures. Polar



(a) branching: 5.4× lower (b) hmm: 105× lower (c) load: 24.9× lower

Fig. 10. Empirical estimator variance $\mathrm{Var}(\hat{\mu}_N)$ versus mean sampling time for the source program P and its determinization $\det P$, on log–log axes. Each point is a sample size $N \in \{10^3, 3 \cdot 10^3, 10^4, 3 \cdot 10^4, 10^5\}$; the estimator variance is the variance of the Monte Carlo estimates across 50 independent replicates. Error bars show ±1 approximate standard error, using $s^2\sqrt{2/49}$ for the sample variance s^2 of the 50 estimates. Subcaptions give the reduction in estimator variance at $N = 10^5$.

builds off earlier work automatically deriving exact moment-based invariants for the more restricted class of Prob-solvable loops [7]; Mora implements this analysis for expected values, variances, and higher moments [8]. Generating function systems such as Genfer, Prodigy, and Geni provide exact moments, including some forms of infinite-state behavior, but operate over more restricted first-order or non-recursive languages [37, 42, 56]. Expectation Programming instead constructs target-aware approximate inference procedures specialized to *estimating* a program's expected return [49]. In contrast to these tools, determinization preserves only the expectation rather than the full output distribution, enabling it to apply to a richer language with a *combination* of higher-order functions, unbounded recursion, recursive data structures, and continuous distributions, enabling the calculation of an *exact* expectation when the determinized program is finite-state.

Quantitative Verification and Invariant Synthesis. A complementary line of work reasons about expectations through quantitative invariants and related proof certificates. McIver and Morgan's weakest pre-expectation calculus provides a deductive framework for expectation-based reasoning, but analyzing loops generally requires supplying suitable invariants [44]. This process can be partially automated using templates and constraint solving to synthesize linear quantitative invariants [36], or through abstract interpretation to infer linear expectation invariants [18]. Subsequent approaches synthesize restricted classes of polynomial invariants [22, 27], or construct martingales whose expectations can be related at termination using optional stopping [5]. Related techniques synthesize supermartingales to bound higher moments of program runtimes and derive tail bounds [40], or combine martingales with concentration inequalities to establish probabilistic bounds [17]. Chatterjee et al. automatically refute equivalence and similarity of output distributions by synthesizing a distinguishing function together with upper expectation supermartingale and lower expectation submartingale certificates [20]. Weakest-preexpectation calculi have also been used to reason about expected runtimes [33] and verify general inequalities over expected values and probabilities [10], with dedicated proof rules for lower bounds on expectations and runtimes [31]. For higher-order programs, [3] translates probabilistic programs with unbounded recursion into deterministic CPS terms denoting their expected cost, and use a higher-order logic to prove upper bounds (no continuous distributions).

There are several other works that establish sound upper or lower bounds on quantitative properties of probabilistic programs, as well. Invariant expectations and upper or lower bounds can be learned from program executions without user-provided templates [4]. Probabilistic affine forms, higher moments, and concentration inequalities can similarly be used to automatically derive bounds for nonlinear probabilistic computations [13]. Moreover, weakest pre-expectations, expected costs, moments, and conditional expectations for higher-order probabilistic programs with continuous distributions [39]. Other approaches automatically derive symbolic upper bounds on expected resource consumption [47]. Chatterjee et al. derive bounds on expected total resource usage and on the probability that it exceeds a given threshold [19]. Further work derives bounds on variance and higher central moments of probabilistic cost accumulators with recursive procedures [54]. Classical work on probabilistic recurrence relations derives concentration bounds for randomized algorithms [35]. Finally, GuBPI computes guaranteed lower and upper bounds on posterior probabilities for higher-order recursive programs with continuous distributions [11] In contrast, our approach is *fully automatic* and computes the *exact expected return* of the program, rather than bounds, when the determinized program is finite-state.

Variance Reduction and Randomness Elimination. Our approach is closely related to conditional Monte Carlo and Rao–Blackwellization, which replace random quantities by conditional expectations to reduce estimator variance [15, 48]. Delayed sampling automatically postpones draws and analytically marginalizes variables when conjugate or affine relationships permit, reducing variance during inference [46]. For them however, the effectiveness can be limited by recursion for example. Subsequent work performs delayed sampling statically when possible, while retaining a dynamic mechanism for programs whose dependencies cannot be resolved at compile time [16]. Semi-symbolic inference similarly combines symbolic manipulation with sampling, performing exact inference for closed distribution families and falling back to particle-based approximation otherwise [2]; inference plans further allow programmers to control which variables are represented symbolically or sampled [23]. In contrast, determinization uses a type-directed source-to-source transformation to replace eligible samples by their means, without requiring conjugacy or symbolic marginalization, while preserving the expected output rather than its full distribution.

Type-Directed Program Transformations. Several probabilistic programming systems use static analyses to guide program transformations. Slice uses a type-directed analysis to discretize continuous randomness [55], while GradInf uses information-flow typing to factor and partially evaluate randomness when constructing gradient estimators [1]. ADEV similarly transforms higher-order probabilistic programs into programs producing unbiased gradient estimates [41]. Determinization follows this type-directed transformation strategy, but uses static dependence information to identify random choices that can be replaced by their expectations.

Data-Availability Statement

An anonymized copy of the artifact is provided as supplementary material. It contains the Lean formalization and the command-line tool, the benchmarks, scripts and recorded results of Section 7, the test suites, and the simulator. A website included in it shows the simulator, the API documentation of the formalization and all source files, and opens in a browser without installing anything. Running `lake build --wfail` checks every proof, and checks that each theorem in `Theorems.lean`, which states the theorems of this paper, uses only Lean's standard axioms (`propext`, `Classical.choice`, `Quot.sound`). The first build takes about half an hour on a 16-core machine. Limitations: the exact results also need the Storm model checker, and the Monte Carlo experiments need Pyro. Timings depend on the machine.

References

[1] Gaurav Arya, Mathieu Huot, Moritz Schauer, Alexander K. Lew, and Feras A. Saad. 2026. GradInf: Gradient Estimation as Probabilistic Inference. *Proceedings of the ACM on Programming Languages* 10, PLDI, Article 243 (2026). doi:10.1145/3808321

[2] Eric Atkinson, Charles Yuan, Guillaume Baudart, Louis Mandel, and Michael Carbin. 2022. Semi-Symbolic Inference for Efficient Streaming Probabilistic Programming. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2, Article 184 (2022). doi:10.1145/3563347

[3] Martin Avanzini, Gilles Barthe, and Ugo Dal Lago. 2021. On Continuation-Passing Transformations and Expected Cost Analysis. *Proceedings of the ACM on Programming Languages* 5, ICFP, Article 87 (2021). doi:10.1145/3473592

[4] Jialu Bao, Nitesh Trivedi, Drashti Pathak, Justin Hsu, and Subhajit Roy. 2022. Data-Driven Invariant Learning for Probabilistic Programs. In *Computer Aided Verification (CAV)*. 33–54. doi:10.1007/978-3-031-13185-1_3

[5] Gilles Barthe, Thomas Espitau, Luis María Ferrer Fioriti, and Justin Hsu. 2016. Synthesizing Probabilistic Invariants via Doob's Decomposition. In *Computer Aided Verification (CAV)*. 43–61. doi:10.1007/978-3-319-41528-4_3

[6] Gilles Barthe, Gian Pietro Farina, Marco Gaboardi, Emilio Jesús Gallego Arias, Andy Gordon, Justin Hsu, and Pierre-Yves Strub. 2016. Differentially Private Bayesian Programming. In *Computer and Communications Security (CCS)*. 68–79. doi:10.1145/2976749.2978371

[7] Ezio Bartocci, Laura Kovács, and Miroslav Stankovič. 2019. Automatic Generation of Moment-Based Invariants for Prob-Solvable Loops. In *Automated Technology for Verification and Analysis (ATVA)*. 255–276. doi:10.1007/978-3-030-31784-3_15

[8] Ezio Bartocci, Laura Kovács, and Miroslav Stankovič. 2020. Mora – Automatic Generation of Moment-Based Invariants. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 492–498. doi:10.1007/978-3-030-45190-5_28

[9] Kevin Batz, Mingshuai Chen, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Philipp Schröer. 2021. Latticed k-Induction with an Application to Probabilistic Programs. In *Computer Aided Verification (CAV)*. 524–549. doi:10.1007/978-3-030-81688-9_25

[10] Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. 2021. Relatively Complete Verification of Probabilistic Programs: An Expressive Language for Expectation-Based Reasoning. *Proceedings of the ACM on Programming Languages* 5, POPL (2021). doi:10.1145/3434320

[11] Raven Beutner, C.-H. Luke Ong, and Fabian Zaiser. 2022. Guaranteed Bounds for Posterior Inference in Universal Probabilistic Programming. In *Programming Language Design and Implementation (PLDI)*. 536–551. doi:10.1145/3519939.3523721

[12] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. 2019. Pyro: Deep Universal Probabilistic Programming. *Journal of Machine Learning Research* 20, 28 (2019), 1–6. https://jmlr.org/papers/v20/18-403.html

[13] Olivier Bouissou, Eric Goubault, Sylvie Putot, Aleksandar Chakarov, and Sriram Sankaranarayanan. 2016. Uncertainty Propagation Using Probabilistic Affine Forms and Concentration of Measure Inequalities. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 225–243. doi:10.1007/978-3-662-49674-9_13

[14] Bob Carpenter, Andrew Gelman, Matthew D. Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2017. Stan: A Probabilistic Programming Language. *Journal of Statistical Software* 76, 1 (2017), 1–32. doi:10.18637/jss.v076.i01

[15] George Casella and Christian P. Robert. 1996. Rao-Blackwellisation of Sampling Schemes. *Biometrika* 83, 1 (1996), 81–94. doi:10.1093/biomet/83.1.81

[16] Gizem Caylak, Daniel Lundén, Viktor Senderov, and David Broman. 2024. Statically and Dynamically Delayed Sampling for Typed Probabilistic Programming Languages. In *Software Language Engineering (SLE)*. doi:10.1145/3687997.3695634

[17] Aleksandar Chakarov and Sriram Sankaranarayanan. 2013. Probabilistic Program Analysis with Martingales. In *Computer Aided Verification (CAV)*. 511–526. doi:10.1007/978-3-642-39799-8_34

[18] Aleksandar Chakarov and Sriram Sankaranarayanan. 2014. Expectation Invariants for Probabilistic Program Loops as Fixed Points. In *Static Analysis Symposium (SAS)*. 85–100. doi:10.1007/978-3-319-10936-7_6

[19] Krishnendu Chatterjee, Amir Kafshdar Goharshady, Tobias Meggendorfer, and Đorđe Žikelić. 2024. Quantitative Bounds on Resource Usage of Probabilistic Programs. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (2024). doi:10.1145/3649824

[20] Krishnendu Chatterjee, Ehsan Kafshdar Goharshady, Petr Novotný, and Đorđe Žikelić. 2024. Equivalence and Similarity Refutation for Probabilistic Programs. *Proceedings of the ACM on Programming Languages* 8, PLDI, Article 232 (2024). doi:10.1145/3656462

[21] Krishnendu Chatterjee, Tim Quatmann, Maximilian Schäffeler, Maximilian Weininger, Tobias Winkler, and Daniel Zilken. 2025. Fixed Point Certificates for Reachability and Expected Rewards in MDPs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (Lecture Notes in Computer Science, Vol. 15697)*. 130–151. doi:10.1007/978-

3-031-90653-4_7

[22] Yu-Fang Chen, Chih-Duo Hong, Bow-Yaw Wang, and Lijun Zhang. 2015. Counterexample-Guided Polynomial Loop Invariant Generation by Lagrange Interpolation. In *Computer Aided Verification (CAV)*. 658–674. doi:10.1007/978-3-319-21690-4_44

[23] Ellie Y. Cheng, Eric Atkinson, Guillaume Baudart, Louis Mandel, and Michael Carbin. 2025. Inference Plans for Hybrid Particle Filtering. *Proceedings of the ACM on Programming Languages* 9, POPL (2025). doi:10.1145/3704846

[24] Marco F. Cusumano-Towner, Feras A. Saad, Alexander K. Lew, and Vikash K. Mansinghka. 2019. Gen: A General-Purpose Probabilistic Programming System with Programmable Inference. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. doi:10.1145/3314221.3314642

[25] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings*. Springer-Verlag, Berlin, Heidelberg, 625–635. doi:10.1007/978-3-030-79876-5_37

[26] Shenghua Feng, Mingshuai Chen, Han Su, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Naijun Zhan. 2023. Lower Bounds for Possibly Divergent Probabilistic Programs. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (2023). doi:10.1145/3586051

[27] Yijun Feng, Lijun Zhang, David N. Jansen, Naijun Zhan, and Bican Xia. 2017. Finding Polynomial Loop Invariants for Probabilistic Programs. In *Automated Technology for Verification and Analysis (ATVA)*. 400–416. doi:10.1007/978-3-319-68167-2_26

[28] Timon Gehr, Sasa Misailovic, and Martin Vechev. 2016. PSI: Exact Symbolic Inference for Probabilistic Programs. In *Computer Aided Verification (CAV)*. 62–83. doi:10.1007/978-3-319-41528-4_4

[29] Timon Gehr, Samuel Steffen, and Martin Vechev. 2020. λPSI: Exact Inference for Higher-Order Probabilistic Programs. In *Programming Language Design and Implementation (PLDI)*. doi:10.1145/3385412.3386006

[30] Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. 2014. Probabilistic Programming. In *Future of Software Engineering (FOSE)*. https://www.microsoft.com/en-us/research/publication/probabilistic-programming/

[31] Marcel Hark, Benjamin Lucien Kaminski, Jürgen Giesl, and Joost-Pieter Katoen. 2020. Aiming Low Is Harder: Induction for Lower Bounds in Probabilistic Program Verification. *Proceedings of the ACM on Programming Languages* 4, POPL (2020). doi:10.1145/3371105

[32] Christian Hensel, Sebastian Junges, Joost-Pieter Katoen, Tim Quatmann, and Matthias Volk. 2022. The Probabilistic Model Checker Storm. *International Journal on Software Tools for Technology Transfer* 24, 4 (2022), 589–610. doi:10.1007/s10009-021-00633-z

[33] Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. 2016. Weakest Precondition Reasoning for Expected Run-Times of Probabilistic Programs. In *European Symposium on Programming (ESOP)*. 364–389. doi:10.1007/978-3-662-49498-1_15

[34] Ahmad Karimi, Marcel Moosbrugger, Miroslav Stankovič, Laura Kovács, Ezio Bartocci, and Efstathia Bura. 2022. Distribution Estimation for Probabilistic Loops. In *Quantitative Evaluation of Systems (QEST)*. 26–42. doi:10.1007/978-3-031-16336-4_2

[35] Richard M. Karp. 1994. Probabilistic Recurrence Relations. *J. ACM* 41, 6 (1994), 1136–1150. doi:10.1145/195613.195632

[36] Joost-Pieter Katoen, Annabelle K. McIver, Larissa A. Meinicke, and Carroll C. Morgan. 2010. Linear-Invariant Generation for Probabilistic Programs: Automated Support for Proof-Based Methods. In *Static Analysis Symposium (SAS)*. 390–406. doi:10.1007/978-3-642-15769-1_24

[37] Lutz Klinkenberg, Christian Blumenthal, Mingshuai Chen, Darion Haase, and Joost-Pieter Katoen. 2024. Exact Bayesian Inference for Loopy Probabilistic Programs Using Generating Functions. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1, Article 127 (2024). doi:10.1145/3649844

[38] Dexter Kozen. 1981. Semantics of Probabilistic Programs. *J. Comput. System Sci.* 22, 3 (1981), 328–350. doi:10.1016/0022-0000(81)90036-2

[39] Satoshi Kura and Hiroshi Unno. 2024. Automated Verification of Higher-Order Probabilistic Programs via a Dependent Refinement Type System. *Proceedings of the ACM on Programming Languages* 8, ICFP (2024). doi:10.1145/3674662

[40] Satoshi Kura, Natsuki Urabe, and Ichiro Hasuo. 2019. Tail Probabilities for Randomized Program Runtimes via Martingales for Higher Moments. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 135–153. doi:10.1007/978-3-030-17465-1_8

[41] Alexander K. Lew, Mathieu Huot, Sam Staton, and Vikash K. Mansinghka. 2023. ADEV: Sound Automatic Differentiation of Expected Values of Probabilistic Programs. *Proceedings of the ACM on Programming Languages* 7, POPL, Article 5 (2023). doi:10.1145/3571198

[42] Jianlin Li and Yizhou Zhang. 2025. Compiling with Generating Functions. *Proceedings of the ACM on Programming Languages* 9, ICFP, Article 265 (2025). doi:10.1145/3747534

[43] The mathlib Community. 2020. The Lean Mathematical Library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs* (New Orleans, LA, USA) *(CPP '20)*. ACM. doi:10.1145/3372885.3373824
[44] Annabelle McIver and Carroll Morgan. 2005. *Abstraction, Refinement and Proof for Probabilistic Systems*. Springer. doi:10.1007/b138392
[45] Marcel Moosbrugger, Miroslav Stankovič, Ezio Bartocci, and Laura Kovács. 2022. This Is the Moment for Probabilistic Loops. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2, Article 178 (2022). doi:10.1145/3563341
[46] Lawrence M. Murray, Daniel Lundén, Jan Kudlicka, David Broman, and Thomas B. Schön. 2018. Delayed Sampling and Automatic Rao-Blackwellization of Probabilistic Programs. In *Artificial Intelligence and Statistics (AISTATS) (Proceedings of Machine Learning Research, Vol. 84)*. 1037–1046. https://proceedings.mlr.press/v84/murray18a.html
[47] Van Chan Ngo, Quentin Carbonneaux, and Jan Hoffmann. 2018. Bounded Expectations: Resource Analysis for Probabilistic Programs. In *Programming Language Design and Implementation (PLDI)*. 496–512. doi:10.1145/3192366.3192394
[48] Art B. Owen. 2013. *Monte Carlo Theory, Methods and Examples*. https://artowen.su.domains/mc/ Chapter 8, Section 8.7.
[49] Tim Reichelt, Adam Goliński, Luke Ong, and Tom Rainforth. 2022. Expectation Programming: Adapting Probabilistic Programming Systems to Estimate Expectations Efficiently. In *Uncertainty in Artificial Intelligence (UAI) (Proceedings of Machine Learning Research, Vol. 180)*. 1676–1685. https://proceedings.mlr.press/v180/reichelt22a.html
[50] Tetsuya Sato, Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Justin Hsu. 2019. Formal Verification of Higher-Order Probabilistic Programs: Reasoning about Approximation, Convergence, Bayesian Inference, and Optimization. *Proceedings of the ACM on Programming Languages* 3, POPL (2019). doi:10.1145/3290351
[51] Dmitriy Traytel, Stefan Berghofer, and Tobias Nipkow. 2011. Extending Hindley–Milner Type Inference with Coercive Structural Subtyping. In *Programming Languages and Systems (APLAS) (Lecture Notes in Computer Science, Vol. 7078)*. 89–104. doi:10.1007/978-3-642-25318-8_10
[52] Matthijs Vákár, Ohad Kammar, and Sam Staton. 2019. A Domain Theory for Statistical Probabilistic Programming. *Proceedings of the ACM on Programming Languages* 3, POPL, Article 36 (2019). doi:10.1145/3290349
[53] Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. 2018. An Introduction to Probabilistic Programming. *arXiv preprint arXiv:1809.10756* (2018). https://arxiv.org/abs/1809.10756
[54] Di Wang, Jan Hoffmann, and Thomas Reps. 2021. Central Moment Analysis for Cost Accumulators in Probabilistic Programs. In *Programming Language Design and Implementation (PLDI)*. doi:10.1145/3453483.3454062
[55] Katherine Wu, Jules Jacobs, Kevin Batz, and Alexandra Silva. 2026. Type-Directed Discretization of Probabilistic Programs. (2026). Manuscript.
[56] Fabian Zaiser, Andrzej S. Murawski, and Luke Ong. 2023. Exact Bayesian Inference on Discrete Models via Probability Generating Functions: A Probabilistic Programming Approach. In *Advances in Neural Information Processing Systems*, Vol. 36. https://arxiv.org/abs/2305.17058
[57] Robert Zinkov and Chung-chieh Shan. 2017. Composing Inference Algorithms as Program Transformations. In *Uncertainty in Artificial Intelligence (UAI)*. https://www.auai.org/uai2017/proceedings/papers/163.pdf

A Appendix

B Gaussian Random Walk

The full source program of Fig. 1, from [52], is shown below. `reduce` folds the step function over the requested times, starting from the path that contains only $(0, 0)$.

```
let reduce =
  rec reduce f => fun accumulator => fun xs =>
    match xs with
    | [] =>
        accumulator
    | x :: rest =>
        reduce f (f accumulator x) rest
in

let step =
  fun path => fun nextTime =>
    match path with
```

```
    | [] =>
        []
    | current :: previous =>
        let time = fst current in
        let position = snd current in
        let nextPosition =
          gauss(position, nextTime - time)
        in
        (nextTime, nextPosition) :: path
in

let times =
  1 :: 2 :: 4 :: 5.5 :: 6 :: 6.25 :: []
in

reduce step ((0, 0) :: []) times
```

C Determinization

We give the full structural definition of the transformation in Definition 2.1 [LEAN], covering every constructor in Fig. 2. To keep the rules compact, we abbreviate recursive calls throughout this section by

$$e' = \text{Determinize}(e), \qquad e'_i = \text{Determinize}(e_i).$$

Thus, each primed expression denotes the determinization of its unprimed counterpart. Bound variable names are unchanged. For each distribution mode α, let

$$\alpha' = \begin{cases} \mathsf{M} & \alpha = \mathsf{E}, \\ \mathsf{G} & \alpha = \mathsf{G}, \\ \mathsf{M} & \alpha = \mathsf{M}. \end{cases}$$

Variables and constants.

$$\begin{aligned} \text{Determinize}(x) &= x, & \text{Determinize}(r) &= r \quad (r \in \mathbb{R}), \\ \text{Determinize}(()) &= (), & \text{Determinize}(\mathsf{true}) &= \mathsf{true}, \\ \text{Determinize}(\mathsf{false}) &= \mathsf{false}, & \text{Determinize}(\mathsf{reject}) &= \mathsf{reject}, \\ \text{Determinize}([]) &= []. \end{aligned}$$

Functions, recursion, and control flow.

$$\begin{aligned} \text{Determinize}(\lambda x.e) &= \lambda x.e', \\ \text{Determinize}(\mathsf{rec}\ f\ x.\ e) &= \mathsf{rec}\ f\ x.\ e', \\ \text{Determinize}(e_1\ e_2) &= e'_1\ e'_2, \\ \text{Determinize}(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2) &= \mathsf{let}\ x = e'_1\ \mathsf{in}\ e'_2, \\ \text{Determinize}(\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2) &= \mathsf{if}\ e'_0\ \mathsf{then}\ e'_1\ \mathsf{else}\ e'_2. \end{aligned}$$

Arithmetic and comparisons.

$$\begin{aligned}
\text{Determinize}(-e) &= -e', \\
\text{Determinize}(e_1 + e_2) &= e_1' + e_2', \\
\text{Determinize}(e_1 \cdot e_2) &= e_1' \cdot e_2', \\
\text{Determinize}(e_1 / e_2) &= e_1' / e_2', \\
\text{Determinize}(e_1 < e_2) &= e_1' < e_2'.
\end{aligned}$$

Pairs, sums, and lists.

$$\begin{aligned}
\text{Determinize}(\langle e_1, e_2 \rangle) &= \langle e_1', e_2' \rangle, \\
\text{Determinize}(\mathsf{fst}\, e) &= \mathsf{fst}\, e', \\
\text{Determinize}(\mathsf{snd}\, e) &= \mathsf{snd}\, e', \\
\text{Determinize}(\mathsf{inl}\, e) &= \mathsf{inl}\, e', \\
\text{Determinize}(\mathsf{inr}\, e) &= \mathsf{inr}\, e', \\
\text{Determinize}(e_1 :: e_2) &= e_1' :: e_2'.
\end{aligned}$$

$$\begin{aligned}
&\text{Determinize}(\mathsf{case}\ e_0\ \mathsf{of}\ \mathsf{inl}\, x \Rightarrow e_1 \mid \mathsf{inr}\, y \Rightarrow e_2) = \\
&\qquad \mathsf{case}\ e_0'\ \mathsf{of}\ \mathsf{inl}\, x \Rightarrow e_1' \mid \mathsf{inr}\, y \Rightarrow e_2', \\
&\text{Determinize}(\mathsf{case}\ e_0\ \mathsf{of}\ [] \Rightarrow e_1 \mid x :: xs \Rightarrow e_2) = \\
&\qquad \mathsf{case}\ e_0'\ \mathsf{of}\ [] \Rightarrow e_1' \mid x :: xs \Rightarrow e_2'.
\end{aligned}$$

Distribution expressions. For all $\alpha \in \{\mathsf{G}, \mathsf{E}, \mathsf{M}\}$, the eight primitive distributions transform as follows:

$$\begin{aligned}
\text{Determinize}(\mathsf{uniform}^{\alpha}(e_1, e_2)) &= \mathsf{uniform}^{\alpha'}(e_1', e_2'), \\
\text{Determinize}(\mathsf{gaussian}^{\alpha}(e_1, e_2)) &= \mathsf{gaussian}^{\alpha'}(e_1', e_2'), \\
\text{Determinize}(\mathsf{poisson}^{\alpha}(e)) &= \mathsf{poisson}^{\alpha'}(e'), \\
\text{Determinize}(\mathsf{exponential}^{\alpha}(e)) &= \mathsf{exponential}^{\alpha'}(e'), \\
\text{Determinize}(\mathsf{bernoulli}^{\alpha}(e)) &= \mathsf{bernoulli}^{\alpha'}(e'), \\
\text{Determinize}(\mathsf{discrete}^{\alpha}(e)) &= \mathsf{discrete}^{\alpha'}(e'), \\
\text{Determinize}(\mathsf{beta}^{\alpha}(e_1, e_2)) &= \mathsf{beta}^{\alpha'}(e_1', e_2'), \\
\text{Determinize}(\mathsf{gamma}^{\alpha}(e_1, e_2)) &= \mathsf{gamma}^{\alpha'}(e_1', e_2').
\end{aligned}$$

The target retains each distribution primitive and all its arguments; only E changes to M, and every argument is recursively transformed. Mean evaluation remains an atomic distribution operation, governed by the domain conditions and reduction rule in Fig. 3.

D Determinization Examples

Example D.1 (Full determinization). Consider an E-mode Gaussian whose mean is given by an E-mode uniform distribution. Determinization changes the outer Gaussian to M-mode and recursively transforms its arguments. In particular, the nested E-mode uniform distribution is also changed to

M-mode:

$$\begin{aligned}\text{Determinize}(\text{gaussian}^{\mathsf{E}}(\text{uniform}^{\mathsf{E}}(0,1),1)) &= \text{gaussian}^{\mathsf{M}}(\text{Determinize}(\text{uniform}^{\mathsf{E}}(0,1)),1)\\ &= \text{gaussian}^{\mathsf{M}}(\text{uniform}^{\mathsf{M}}(0,1),1).\end{aligned}$$

Thus, both sampling sites are replaced by mean evaluations, yielding a deterministic program.

Example D.2 (Partial determinization). Consider a program with dependent random draws. The G-mode uniform distribution remains stochastic, while the E-mode Gaussian is changed to M-mode. Crucially, the Gaussian's mean $2x + 1$ still depends on the random value of x:

$$\text{Determinize}\left(\begin{array}{l}\text{let } x = \text{uniform}^{\mathsf{G}}(0,1) \text{ in}\\ \text{let } y = \text{gaussian}^{\mathsf{E}}(2x+1,\sigma^2) \text{ in}\\ x * y\end{array}\right) = \begin{array}{l}\text{let } x = \text{uniform}^{\mathsf{G}}(0,1) \text{ in}\\ \text{let } y = \text{gaussian}^{\mathsf{M}}(2x+1,\sigma^2) \text{ in}\\ x * y.\end{array}$$

Thus, determinization eliminates the Gaussian's conditional randomness while preserving its dependence on the retained random value of x.

E Typing Rules

Type grammar, real typing, and subtyping rules. For distribution modes $\alpha \in \{\mathsf{G}, \mathsf{E}\}$, the type annotation m must match α; for $\alpha = \mathsf{M}$, either type annotation is permitted. We omit the standard rules for variables, unit and Boolean literals, functions, recursion, application, let bindings, conditionals, pairs, projections, sum injections and case analysis, and list constructors and case analysis.

Type grammar.

$$\tau ::= \text{unit} \mid \text{bool} \mid \text{real}^m \mid \tau \to \tau \mid \tau \times \tau \mid \tau + \tau \mid \text{list}(\tau) \qquad m ::= \mathsf{G} \mid \mathsf{E}$$

Real typing.

$$\frac{r \in \mathbb{R}}{\Gamma \vdash r : \text{real}^m}\ \textsc{Real} \qquad \frac{}{\Gamma \vdash \text{reject} : \tau}\ \textsc{Reject} \qquad \frac{\Gamma \vdash e : \text{real}^m}{\Gamma \vdash -e : \text{real}^m}\ \textsc{Neg}$$

$$\frac{\Gamma \vdash e_1 : \text{real}^m \qquad \Gamma \vdash e_2 : \text{real}^m}{\Gamma \vdash e_1 + e_2 : \text{real}^m}\ \textsc{Add} \qquad \frac{\Gamma \vdash e_1 : \text{real}^{\mathsf{G}} \qquad \Gamma \vdash e_2 : \text{real}^m}{\Gamma \vdash e_1 \cdot e_2 : \text{real}^m}\ \textsc{Mul}$$

$$\frac{\Gamma \vdash e_1 : \text{real}^m \qquad \Gamma \vdash e_2 : \text{real}^{\mathsf{G}}}{\Gamma \vdash e_1 / e_2 : \text{real}^m}\ \textsc{Div} \qquad \frac{\Gamma \vdash e_1 : \text{real}^{\mathsf{G}} \qquad \Gamma \vdash e_2 : \text{real}^{\mathsf{G}}}{\Gamma \vdash e_1 < e_2 : \text{bool}}\ \textsc{Cmp}$$

$$\frac{\Gamma \vdash a : \mathsf{real}^m \qquad \Gamma \vdash b : \mathsf{real}^m \qquad \alpha = \mathsf{M} \vee \alpha = m}{\Gamma \vdash \mathsf{uniform}^\alpha(a, b) : \mathsf{real}^m}\ \textsc{Uniform}$$

$$\frac{\Gamma \vdash u : \mathsf{real}^m \qquad \Gamma \vdash v : \mathsf{real}^{\mathsf{G}} \qquad \alpha = \mathsf{M} \vee \alpha = m}{\Gamma \vdash \mathsf{gaussian}^\alpha(u, v) : \mathsf{real}^m}\ \textsc{Gaussian}$$

$$\frac{\Gamma \vdash \lambda : \mathsf{real}^m \qquad \alpha = \mathsf{M} \vee \alpha = m}{\Gamma \vdash \mathsf{poisson}^\alpha(\lambda) : \mathsf{real}^m}\ \textsc{Poisson} \qquad \frac{\Gamma \vdash \lambda : \mathsf{real}^{\mathsf{G}} \qquad \alpha = \mathsf{M} \vee \alpha = m}{\Gamma \vdash \mathsf{exponential}^\alpha(\lambda) : \mathsf{real}^m}\ \textsc{Exponential}$$

$$\frac{\Gamma \vdash p : \mathsf{real}^m \qquad \alpha = \mathsf{M} \vee \alpha = m}{\Gamma \vdash \mathsf{bernoulli}^\alpha(p) : \mathsf{real}^m}\ \textsc{Bernoulli} \qquad \frac{\Gamma \vdash p : \mathsf{list}(\mathsf{real}^m) \qquad \alpha = \mathsf{M} \vee \alpha = m}{\Gamma \vdash \mathsf{discrete}^\alpha(p) : \mathsf{real}^m}\ \textsc{Discrete}$$

$$\frac{\Gamma \vdash a : \mathsf{real}^{\mathsf{G}} \qquad \Gamma \vdash b : \mathsf{real}^{\mathsf{G}} \qquad \alpha = \mathsf{M} \vee \alpha = m}{\Gamma \vdash \mathsf{beta}^\alpha(a, b) : \mathsf{real}^m}\ \textsc{Beta}$$

$$\frac{\Gamma \vdash k : \mathsf{real}^m \qquad \Gamma \vdash \lambda : \mathsf{real}^{\mathsf{G}} \qquad \alpha = \mathsf{M} \vee \alpha = m}{\Gamma \vdash \mathsf{gamma}^\alpha(k, \lambda) : \mathsf{real}^m}\ \textsc{Gamma}$$

Subtyping.

$$\frac{}{\tau <: \tau}\ \textsc{Refl} \qquad \frac{\tau <: \sigma \qquad \sigma <: \rho}{\tau <: \rho}\ \textsc{Trans} \qquad \frac{}{\mathsf{real}^{\mathsf{G}} <: \mathsf{real}^{\mathsf{E}}}\ \textsc{Real-Sub}$$

$$\frac{\sigma_1 <: \tau_1 \qquad \tau_2 <: \sigma_2}{\tau_1 \to \tau_2 <: \sigma_1 \to \sigma_2}\ \textsc{Arrow} \qquad \frac{\tau_1 <: \sigma_1 \qquad \tau_2 <: \sigma_2}{\tau_1 \times \tau_2 <: \sigma_1 \times \sigma_2}\ \textsc{Product} \qquad \frac{\tau_1 <: \sigma_1 \qquad \tau_2 <: \sigma_2}{\tau_1 + \tau_2 <: \sigma_1 + \sigma_2}\ \textsc{Sum}$$

$$\frac{\tau <: \sigma}{\mathsf{list}(\tau) <: \mathsf{list}(\sigma)}\ \textsc{List} \qquad \frac{\Gamma \vdash e : \tau \qquad \tau <: \sigma}{\Gamma \vdash e : \sigma}\ \textsc{Sub}$$

F Constraint Generation Rules

We give the full constraint-generation rules for every construct in Fig. 2, including those selected for Fig. 5. The judgment $\Gamma \vdash e \Rightarrow \tau \mid C$ infers a type τ and the complete constraint set C. Each rule explicitly includes the constraints generated by its subexpressions and any additional requirements. Type and affinity variables declared fresh are distinct from all variables already in use. Bindings are monomorphic.

Compact summary. The following table summarizes constraint generation for arithmetic expressions and distributions. For each operand, $\Gamma \vdash e_i \Rightarrow \tau_i \mid C_i$. Each displayed a is fresh; the complete constraint set is $\bigcup_i C_i$ together with the listed constraints. At a sampling site, a supplies the inferred annotation; an explicit annotation $m \in \{\mathsf{G}, \mathsf{E}\}$ additionally generates $a = m$.

Expression	Inferred type	Added constraints
r	real^a	$\varnothing$
$-e_1$	real^a	$\tau_1 <: \text{real}^a$
$e_1 + e_2$	real^a	$\tau_1 <: \text{real}^a, \quad \tau_2 <: \text{real}^a$
$e_1 \cdot e_2$	real^a	$\tau_1 <: \text{real}^{\text{G}}, \quad \tau_2 <: \text{real}^a$
e_1 / e_2	real^a	$\tau_1 <: \text{real}^a, \quad \tau_2 <: \text{real}^{\text{G}}$
$e_1 < e_2$	bool	$\tau_1 <: \text{real}^{\text{G}}, \quad \tau_2 <: \text{real}^{\text{G}}$
$\text{uniform}(e_1, e_2)$	real^a	$\tau_1 <: \text{real}^a, \quad \tau_2 <: \text{real}^a$
$\text{gaussian}(e_1, e_2)$	real^a	$\tau_1 <: \text{real}^a, \quad \tau_2 <: \text{real}^{\text{G}}$
$\text{discrete}(e_1)$	real^a	$\tau_1 <: \text{list}(\text{real}^a)$

Variables and constants.

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x \Rightarrow \tau \mid \varnothing}\ \textsc{Var}$$

$$\frac{r \in \mathbb{R} \qquad a \text{ fresh}}{\Gamma \vdash r \Rightarrow \text{real}^a \mid \varnothing}\ \textsc{Real}$$

$$\frac{}{\Gamma \vdash () \Rightarrow \text{unit} \mid \varnothing}\ \textsc{Unit}$$

$$\frac{}{\Gamma \vdash \text{true} \Rightarrow \text{bool} \mid \varnothing}\ \textsc{True}$$

$$\frac{}{\Gamma \vdash \text{false} \Rightarrow \text{bool} \mid \varnothing}\ \textsc{False}$$

$$\frac{X \text{ fresh}}{\Gamma \vdash \text{reject} \Rightarrow X \mid \varnothing}\ \textsc{Reject}$$

Functions, bindings, and conditionals.

$$\frac{X \text{ fresh} \qquad \Gamma, x : X \vdash e \Rightarrow \tau \mid C}{\Gamma \vdash \lambda x.e \Rightarrow X \to \tau \mid C}\ \textsc{Abs}$$

$$\frac{X, Y \text{ fresh} \qquad \Gamma, f : X \to Y, x : X \vdash e \Rightarrow \tau \mid C}{\Gamma \vdash \text{rec } f\ x.\, e \Rightarrow X \to Y \mid C \cup \{\tau <: Y\}}\ \textsc{Rec}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \mid C_1 \qquad \Gamma \vdash e_2 \Rightarrow \tau_2 \mid C_2 \qquad X, Y \text{ fresh}}{\Gamma \vdash e_1\, e_2 \Rightarrow Y \mid C_1 \cup C_2 \cup \{\tau_1 <: (X \to Y),\ \tau_2 <: X\}}\ \textsc{App}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \mid C_1 \qquad \Gamma, x : \tau_1 \vdash e_2 \Rightarrow \tau_2 \mid C_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 \Rightarrow \tau_2 \mid C_1 \cup C_2}\ \textsc{Let}$$

$$\frac{\Gamma \vdash e_0 \Rightarrow \tau_0 \mid C_0 \qquad \Gamma \vdash e_1 \Rightarrow \tau_1 \mid C_1 \qquad \Gamma \vdash e_2 \Rightarrow \tau_2 \mid C_2 \qquad X \text{ fresh}}{\Gamma \vdash \text{if } e_0 \text{ then } e_1 \text{ else } e_2 \Rightarrow X \mid C_0 \cup C_1 \cup C_2 \cup \{\tau_0 <: \text{bool},\ \tau_1 <: X,\ \tau_2 <: X\}}\ \textsc{If}$$

Arithmetic and comparisons.

$$\frac{\Gamma \vdash e \Rightarrow \tau \mid C \qquad a \text{ fresh}}{\Gamma \vdash -e \Rightarrow \text{real}^a \mid C \cup \{\tau <: \text{real}^a\}}\ \textsc{Neg}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \mid C_1 \qquad \Gamma \vdash e_2 \Rightarrow \tau_2 \mid C_2 \qquad a \text{ fresh}}{\Gamma \vdash e_1 + e_2 \Rightarrow \text{real}^a \mid C_1 \cup C_2 \cup \{\tau_1 <: \text{real}^a,\ \tau_2 <: \text{real}^a\}}\ \textsc{Add}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \mid C_1 \qquad \Gamma \vdash e_2 \Rightarrow \tau_2 \mid C_2 \qquad a \text{ fresh}}{\Gamma \vdash e_1 \cdot e_2 \Rightarrow \text{real}^a \mid C_1 \cup C_2 \cup \{\tau_1 <: \text{real}^{\mathrm{G}},\ \tau_2 <: \text{real}^a\}}\ \textsc{Mul}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \mid C_1 \qquad \Gamma \vdash e_2 \Rightarrow \tau_2 \mid C_2 \qquad a \text{ fresh}}{\Gamma \vdash e_1 / e_2 \Rightarrow \text{real}^a \mid C_1 \cup C_2 \cup \{\tau_1 <: \text{real}^a,\ \tau_2 <: \text{real}^{\mathrm{G}}\}}\ \textsc{Div}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \mid C_1 \qquad \Gamma \vdash e_2 \Rightarrow \tau_2 \mid C_2}{\Gamma \vdash e_1 < e_2 \Rightarrow \text{bool} \mid C_1 \cup C_2 \cup \{\tau_1 <: \text{real}^{\mathrm{G}},\ \tau_2 <: \text{real}^{\mathrm{G}}\}}\ \textsc{Cmp}$$

Products and sums.

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \mid C_1 \qquad \Gamma \vdash e_2 \Rightarrow \tau_2 \mid C_2}{\Gamma \vdash \langle e_1, e_2 \rangle \Rightarrow \tau_1 \times \tau_2 \mid C_1 \cup C_2}\ \textsc{Pair}$$

$$\frac{\Gamma \vdash e \Rightarrow \tau \mid C \qquad X, Y \text{ fresh}}{\Gamma \vdash \text{fst}\, e \Rightarrow X \mid C \cup \{\tau <: X \times Y\}}\ \textsc{Fst}$$

$$\frac{\Gamma \vdash e \Rightarrow \tau \mid C \qquad X, Y \text{ fresh}}{\Gamma \vdash \text{snd}\, e \Rightarrow Y \mid C \cup \{\tau <: X \times Y\}}\ \textsc{Snd}$$

$$\frac{\Gamma \vdash e \Rightarrow \tau \mid C \qquad X \text{ fresh}}{\Gamma \vdash \text{inl}\, e \Rightarrow \tau + X \mid C}\ \textsc{Inl}$$

$$\frac{\Gamma \vdash e \Rightarrow \tau \mid C \qquad X \text{ fresh}}{\Gamma \vdash \text{inr}\, e \Rightarrow X + \tau \mid C}\ \textsc{Inr}$$

$$\frac{\Gamma \vdash e_0 \Rightarrow \tau_0 \mid C_0 \qquad \Gamma, x : X \vdash e_1 \Rightarrow \tau_1 \mid C_1 \qquad \Gamma, y : Y \vdash e_2 \Rightarrow \tau_2 \mid C_2 \qquad X, Y, Z \text{ fresh}}{\Gamma \vdash \text{case } e_0 \text{ of inl } x \Rightarrow e_1 \mid \text{inr } y \Rightarrow e_2 \Rightarrow Z \mid C_0 \cup C_1 \cup C_2 \cup \{\tau_0 <: X + Y,\ \tau_1 <: Z,\ \tau_2 <: Z\}}\ \textsc{Sum-Case}$$

Lists.

$$\frac{X \text{ fresh}}{\Gamma \vdash [] \Rightarrow \mathsf{list}(X) \mid \varnothing}\ \textsc{Nil}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \mid C_1 \qquad \Gamma \vdash e_2 \Rightarrow \tau_2 \mid C_2 \qquad X \text{ fresh}}{\Gamma \vdash e_1 :: e_2 \Rightarrow \mathsf{list}(X) \mid C_1 \cup C_2 \cup \{\tau_1 <: X,\ \tau_2 <: \mathsf{list}(X)\}}\ \textsc{Cons}$$

$$\frac{\Gamma \vdash e_0 \Rightarrow \tau_0 \mid C_0 \qquad \Gamma \vdash e_1 \Rightarrow \tau_1 \mid C_1 \qquad \Gamma, x : X, xs : \mathsf{list}(X) \vdash e_2 \Rightarrow \tau_2 \mid C_2 \qquad X, Z \text{ fresh}}{\Gamma \vdash \mathsf{case}\ e_0\ \mathsf{of}\ [] \Rightarrow e_1 \mid x :: xs \Rightarrow e_2 \Rightarrow Z \mid C_0 \cup C_1 \cup C_2 \cup \{\tau_0 <: \mathsf{list}(X),\ \tau_1 <: Z,\ \tau_2 <: Z\}}\ \textsc{List-Case}$$

Distributions. To cover both inferred and explicit annotations, let $\alpha = \circ$ denote an omitted sampling annotation (an inference-input placeholder), and define the annotation constraints by

$$A(\alpha, a) = \begin{cases} \{a = m\} & \alpha = m \in \{\mathsf{G}, \mathsf{E}\}, \\ \varnothing & \alpha \in \{\circ, \mathsf{M}\}. \end{cases}$$

At a site with $\alpha = \circ$, the solved value of a supplies the sampling annotation. Explicit sampling annotations stay fixed; mean-evaluation annotations remain M. Each rule below includes $A(\alpha, a)$ in its complete constraint set.

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \mid C_1 \qquad \Gamma \vdash e_2 \Rightarrow \tau_2 \mid C_2 \qquad a \text{ fresh}}{\Gamma \vdash \mathsf{uniform}^{\alpha}(e_1, e_2) \Rightarrow \mathsf{real}^{a} \mid C_1 \cup C_2 \cup A(\alpha, a) \cup \{\tau_1 <: \mathsf{real}^{a},\ \tau_2 <: \mathsf{real}^{a}\}}\ \textsc{Uniform}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \mid C_1 \qquad \Gamma \vdash e_2 \Rightarrow \tau_2 \mid C_2 \qquad a \text{ fresh}}{\Gamma \vdash \mathsf{gaussian}^{\alpha}(e_1, e_2) \Rightarrow \mathsf{real}^{a} \mid C_1 \cup C_2 \cup A(\alpha, a) \cup \{\tau_1 <: \mathsf{real}^{a},\ \tau_2 <: \mathsf{real}^{\mathsf{G}}\}}\ \textsc{Gaussian}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \mid C_1 \qquad \Gamma \vdash e_2 \Rightarrow \tau_2 \mid C_2 \qquad a \text{ fresh}}{\Gamma \vdash \mathsf{beta}^{\alpha}(e_1, e_2) \Rightarrow \mathsf{real}^{a} \mid C_1 \cup C_2 \cup A(\alpha, a) \cup \{\tau_1 <: \mathsf{real}^{\mathsf{G}},\ \tau_2 <: \mathsf{real}^{\mathsf{G}}\}}\ \textsc{Beta}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \mid C_1 \qquad \Gamma \vdash e_2 \Rightarrow \tau_2 \mid C_2 \qquad a \text{ fresh}}{\Gamma \vdash \mathsf{gamma}^{\alpha}(e_1, e_2) \Rightarrow \mathsf{real}^{a} \mid C_1 \cup C_2 \cup A(\alpha, a) \cup \{\tau_1 <: \mathsf{real}^{a},\ \tau_2 <: \mathsf{real}^{\mathsf{G}}\}}\ \textsc{Gamma}$$

$$\frac{\Gamma \vdash e \Rightarrow \tau \mid C \qquad a \text{ fresh}}{\Gamma \vdash \mathsf{poisson}^{\alpha}(e) \Rightarrow \mathsf{real}^{a} \mid C \cup A(\alpha, a) \cup \{\tau <: \mathsf{real}^{a}\}}\ \textsc{Poisson}$$

$$\frac{\Gamma \vdash e \Rightarrow \tau \mid C \qquad a \text{ fresh}}{\Gamma \vdash \mathsf{exponential}^{\alpha}(e) \Rightarrow \mathsf{real}^{a} \mid C \cup A(\alpha, a) \cup \{\tau <: \mathsf{real}^{\mathsf{G}}\}}\ \textsc{Exponential}$$

$$\frac{\Gamma \vdash e \Rightarrow \tau \mid C \qquad a \text{ fresh}}{\Gamma \vdash \mathsf{bernoulli}^{\alpha}(e) \Rightarrow \mathsf{real}^{a} \mid C \cup A(\alpha, a) \cup \{\tau <: \mathsf{real}^{a}\}}\ \textsc{Bernoulli}$$

$$\frac{\Gamma \vdash e \Rightarrow \tau \mid C \qquad a \text{ fresh}}{\Gamma \vdash \mathsf{discrete}^{\alpha}(e) \Rightarrow \mathsf{real}^{a} \mid C \cup A(\alpha, a) \cup \{\tau <: \mathsf{list}(\mathsf{real}^{a})\}}\ \textsc{Discrete}$$

G Constraint Generation Example

Example G.1 (Constraint generation). Consider the following program, with all sampling sites unannotated:

$$\begin{aligned}&\text{let } x = \text{uniform}(0,1) \text{ in}\\ &\text{let } y = \text{gaussian}(x,1) \text{ in}\\ &\text{let } z = \text{gaussian}(y,1) \text{ in}\\ &\text{if } x < 1/2 \text{ then } x * z \text{ else } z + 1.\end{aligned}$$

Let a_x, a_y, a_z denote the fresh mode variables associated with the three sampling sites. By Let and Var, subsequent uses of x, y, z have types real^{a_x}, real^{a_y}, real^{a_z}, respectively. Similarly, applying the remaining rules in Fig. 5 generates the following subtype constraints:

$$\begin{aligned}\text{real}^{a_x} <: \text{real}^{a_y}, \quad \text{real}^{a_y} <: \text{real}^{a_z} \quad &\text{(Gaussian)}\\ \text{real}^{a_x} <: \text{real}^{\mathsf{G}} \quad &\text{(Cmp, Mul)}\\ \text{real}^{a_z} <: \text{real}^{a_t}, \quad \text{real}^{a_z} <: \text{real}^{a_f} \quad &\text{(Mul, Add)}\\ \text{real}^{a_t} <: X, \quad \text{real}^{a_f} <: X \quad &\text{(If)}.\end{aligned}$$

Here a_f is the mode variable of the else branch's result type, and X is the fresh type variable introduced by If. We omit the constraints involving numeric constants and the trivial Boolean guard constraint; these are generated by the same rules.

H Constraint Solving Example

Example H.1 (Constraint solving). Continuing Example G.1, we apply Fig. 6. Shape unification succeeds and gives X the shape of a real type. Decoration therefore replaces every occurrence of X by real^b with the same fresh annotation variable b. Subtype constraint reduction turns $\text{real}^{a_t} <: \text{real}^b$ and $\text{real}^{a_f} <: \text{real}^b$ into $a_t \le b$ and $a_f \le b$.

The remaining real-type constraints reduce directly to mode inequalities:

$$\begin{aligned}a_x \le a_y, \quad a_y \le a_z, \quad a_x \le \mathsf{G},\\ a_z \le a_t, \quad a_z \le a_f, \quad a_t \le b, \quad a_f \le b.\end{aligned}$$

Mode constraint solving forces $a_x = \mathsf{G}$, since $a_x \le \mathsf{G}$. No other mode variables are forced to G, so all remaining variables are assigned E. Thus, inference assigns the sampling sites

$$(a_x, a_y, a_z) = (\mathsf{G}, \mathsf{E}, \mathsf{E})$$

and the program has type real^{E}. The inferred annotations yield the program:

$$\begin{aligned}&\text{let } x = \text{uniform}^{\mathsf{G}}(0,1) \text{ in}\\ &\text{let } y = \text{gaussian}^{\mathsf{E}}(x,1) \text{ in}\\ &\text{let } z = \text{gaussian}^{\mathsf{E}}(y,1) \text{ in}\\ &\text{if } x < 1/2 \text{ then } x * z \text{ else } z + 1.\end{aligned}$$

Determinization consequently retains the uniform sample while replacing both Gaussian samples by their conditional means.

I Tracewise Soundness Theorem, by Example

Example I.1. Consider

$$\begin{aligned}&\text{let } x = \text{uniform}^{\mathsf{G}}(0,1) \text{ in}\\ &\text{let } y = \text{uniform}^{\mathsf{E}}(-1,1) \text{ in } x + y.\end{aligned}$$

A trace records only the sampled value $x = r$. Given this trace, the original program returns $r + y$, which is uniformly distributed on $[r-1, r+1]$ and has expectation r. Determinization replaces the sample y by its mean, zero, so the transformed program returns r. Thus, for each such trace τ,

$$\mathbb{E}[e \mid \tau] = r, \qquad \mu_{\mathrm{Determinize}(e)}(\cdot \mid \tau) = \delta_r.$$

The output still varies across traces, but within each trace determinization replaces the remaining randomness by its average.

J Symbolic Semantics

J.1 Residual Expression Syntax

Definition J.1 (Residual symbolic expressions). Fix symbolic variables $u_1, \ldots, u_k$. Affine expressions and residual symbolic expressions are given by the following grammar:

$$\begin{aligned}
a &::= c_0 + \sum_{i=1}^{k} c_i u_i, \qquad c_0, \ldots, c_k \in \mathbb{R}, \\
\widehat{e} &::= x \mid \mathsf{sym}(a) \mid () \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{reject} \\
&\quad \mid \lambda x.\widehat{e} \mid \mathsf{rec}\ f\ x.\widehat{e} \mid \widehat{e}\,\widehat{e} \mid \mathsf{let}\ x = \widehat{e}\ \mathsf{in}\ \widehat{e} \\
&\quad \mid \mathsf{if}\ \widehat{e}\ \mathsf{then}\ \widehat{e}\ \mathsf{else}\ \widehat{e} \mid -\widehat{e} \mid \widehat{e} + \widehat{e} \mid \widehat{e} \cdot \widehat{e} \mid \widehat{e}/\widehat{e} \mid \widehat{e} < \widehat{e} \\
&\quad \mid \langle \widehat{e}, \widehat{e} \rangle \mid \mathsf{fst}\,\widehat{e} \mid \mathsf{snd}\,\widehat{e} \\
&\quad \mid \mathsf{inl}\,\widehat{e} \mid \mathsf{inr}\,\widehat{e} \mid \mathsf{case}\ \widehat{e}\ \mathsf{of}\ \mathsf{inl}\,x \Rightarrow \widehat{e}_1 \mid \mathsf{inr}\,y \Rightarrow \widehat{e}_2 \\
&\quad \mid [] \mid \widehat{e} :: \widehat{e} \mid \mathsf{case}\ \widehat{e}\ \mathsf{of}\ [] \Rightarrow \widehat{e}_0 \mid x :: xs \Rightarrow \widehat{e}_1 \\
&\quad \mid \mathsf{uniform}^{\alpha}(\widehat{e}, \widehat{e}) \mid \mathsf{gaussian}^{\alpha}(\widehat{e}, \widehat{e}) \mid \mathsf{poisson}^{\alpha}(\widehat{e}) \mid \mathsf{exponential}^{\alpha}(\widehat{e}) \\
&\quad \mid \mathsf{bernoulli}^{\alpha}(\widehat{e}) \mid \mathsf{discrete}^{\alpha}(\widehat{e}) \mid \mathsf{beta}^{\alpha}(\widehat{e}, \widehat{e}) \mid \mathsf{gamma}^{\alpha}(\widehat{e}, \widehat{e}), \\
\alpha &::= \mathsf{G} \mid \mathsf{E} \mid \mathsf{M}.
\end{aligned}$$

Thus, the residual grammar is the source grammar with each real literal replaced by a wrapped affine expression $\mathsf{sym}(a)$. Symbolic coordinates u_i are distinct from program variables bound by functions and pattern matching. Substitution for program variables is capture avoiding.

Values and evaluation contexts. Residual values and left-to-right evaluation contexts are:

$$\begin{aligned}
\widehat{v} &::= \mathsf{sym}(a) \mid () \mid \mathsf{true} \mid \mathsf{false} \mid \lambda x.\widehat{e} \mid \mathsf{rec}\ f\ x.\widehat{e} \\
&\quad \mid \langle \widehat{v}, \widehat{v} \rangle \mid \mathsf{inl}\,\widehat{v} \mid \mathsf{inr}\,\widehat{v} \mid [] \mid \widehat{v} :: \widehat{v}, \\
\widehat{C} &::= [\,] \mid \widehat{C}\,\widehat{e} \mid \widehat{v}\,\widehat{C} \mid \mathsf{let}\ x = \widehat{C}\ \mathsf{in}\ \widehat{e} \mid \mathsf{if}\ \widehat{C}\ \mathsf{then}\ \widehat{e}_1\ \mathsf{else}\ \widehat{e}_2 \\
&\quad \mid \langle \widehat{C}, \widehat{e} \rangle \mid \langle \widehat{v}, \widehat{C} \rangle \mid \mathsf{fst}\,\widehat{C} \mid \mathsf{snd}\,\widehat{C} \mid \mathsf{inl}\,\widehat{C} \mid \mathsf{inr}\,\widehat{C} \\
&\quad \mid \widehat{C} :: \widehat{e} \mid \widehat{v} :: \widehat{C} \mid \mathsf{case}\ \widehat{C}\ \mathsf{of}\ \mathsf{inl}\,x \Rightarrow \widehat{e}_1 \mid \mathsf{inr}\,y \Rightarrow \widehat{e}_2 \mid \mathsf{case}\ \widehat{C}\ \mathsf{of}\ [] \Rightarrow \widehat{e}_0 \mid x :: xs \Rightarrow \widehat{e}_1 \\
&\quad \mid -\widehat{C} \mid \widehat{C} \circ \widehat{e} \mid \widehat{v} \circ \widehat{C} \mid \mathsf{D}^{\alpha}(\widehat{v}, \widehat{C}, \widehat{e}),
\end{aligned}$$

where $\circ \in \{+, \cdot, /, <\}$ and $\alpha \in \{\mathsf{G}, \mathsf{E}, \mathsf{M}\}$.

J.2 Residual Expression Typing Rules

Definition J.2 (Typing residual symbolic expressions). Residual symbolic expressions use the type grammar from Fig. 4. The judgment $\Gamma \vdash \widehat{e} : \tau$ is generated by the following rules. The two affine-literal rules are the only rules that differ from the source typing judgment.

$$\frac{a = c_0 + \sum_{i=1}^{k} c_i u_i}{\Gamma \vdash \mathsf{sym}(a) : \mathsf{real}^{\mathrm{E}}}\ \textsc{Affine-E} \qquad \frac{a = c_0 + \sum_{i=1}^{k} c_i u_i \qquad c_1 = \cdots = c_k = 0}{\Gamma \vdash \mathsf{sym}(a) : \mathsf{real}^{\mathrm{G}}}\ \textsc{Affine-G}$$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \textsc{Var} \qquad \frac{}{\Gamma \vdash () : \mathsf{unit}}\ \textsc{Unit} \qquad \frac{}{\Gamma \vdash \mathsf{true} : \mathsf{bool}}\ \textsc{True} \qquad \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{bool}}\ \textsc{False}$$

$$\frac{}{\Gamma \vdash \mathsf{reject} : \tau}\ \textsc{Reject}$$

$$\frac{\Gamma, x : \tau_1 \vdash \widehat{e} : \tau_2}{\Gamma \vdash \lambda x.\widehat{e} : \tau_1 \to \tau_2}\ \textsc{Lam} \qquad \frac{\Gamma, f : \tau_1 \to \tau_2, x : \tau_1 \vdash \widehat{e} : \tau_2}{\Gamma \vdash \mathsf{rec}\ f\ x.\widehat{e} : \tau_1 \to \tau_2}\ \textsc{Rec}$$

$$\frac{\Gamma \vdash \widehat{e}_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash \widehat{e}_2 : \tau_1}{\Gamma \vdash \widehat{e}_1\ \widehat{e}_2 : \tau_2}\ \textsc{App} \qquad \frac{\Gamma \vdash \widehat{e}_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash \widehat{e}_2 : \tau_2}{\Gamma \vdash \mathsf{let}\ x = \widehat{e}_1\ \mathsf{in}\ \widehat{e}_2 : \tau_2}\ \textsc{Let}$$

$$\frac{\Gamma \vdash \widehat{e}_1 : \mathsf{bool} \qquad \Gamma \vdash \widehat{e}_2 : \tau \qquad \Gamma \vdash \widehat{e}_3 : \tau}{\Gamma \vdash \mathsf{if}\ \widehat{e}_1\ \mathsf{then}\ \widehat{e}_2\ \mathsf{else}\ \widehat{e}_3 : \tau}\ \textsc{If}$$

$$\frac{\Gamma \vdash \widehat{e}_1 : \tau_1 \qquad \Gamma \vdash \widehat{e}_2 : \tau_2}{\Gamma \vdash \langle \widehat{e}_1, \widehat{e}_2 \rangle : \tau_1 \times \tau_2}\ \textsc{Pair} \qquad \frac{\Gamma \vdash \widehat{e} : \tau_1 \times \tau_2}{\Gamma \vdash \mathsf{fst}\,\widehat{e} : \tau_1}\ \textsc{Fst} \qquad \frac{\Gamma \vdash \widehat{e} : \tau_1 \times \tau_2}{\Gamma \vdash \mathsf{snd}\,\widehat{e} : \tau_2}\ \textsc{Snd}$$

$$\frac{\Gamma \vdash \widehat{e} : \tau_1}{\Gamma \vdash \mathsf{inl}\,\widehat{e} : \tau_1 + \tau_2}\ \textsc{Inl} \qquad \frac{\Gamma \vdash \widehat{e} : \tau_2}{\Gamma \vdash \mathsf{inr}\,\widehat{e} : \tau_1 + \tau_2}\ \textsc{Inr}$$

$$\frac{\Gamma \vdash \widehat{e} : \tau_1 + \tau_2 \qquad \Gamma, x : \tau_1 \vdash \widehat{e}_1 : \tau \qquad \Gamma, y : \tau_2 \vdash \widehat{e}_2 : \tau}{\Gamma \vdash \mathsf{case}\ \widehat{e}\ \mathsf{of}\ \mathsf{inl}\,x \Rightarrow \widehat{e}_1 \mid \mathsf{inr}\,y \Rightarrow \widehat{e}_2 : \tau}\ \textsc{Sum-Case}$$

$$\frac{}{\Gamma \vdash [] : \mathsf{list}(\tau)}\ \textsc{Nil} \qquad \frac{\Gamma \vdash \widehat{e}_1 : \tau \qquad \Gamma \vdash \widehat{e}_2 : \mathsf{list}(\tau)}{\Gamma \vdash \widehat{e}_1 :: \widehat{e}_2 : \mathsf{list}(\tau)}\ \textsc{Cons}$$

$$\frac{\Gamma \vdash \widehat{e} : \mathsf{list}(\tau_1) \qquad \Gamma \vdash \widehat{e}_0 : \tau_2 \qquad \Gamma, x : \tau_1, xs : \mathsf{list}(\tau_1) \vdash \widehat{e}_1 : \tau_2}{\Gamma \vdash \mathsf{case}\ \widehat{e}\ \mathsf{of}\ [] \Rightarrow \widehat{e}_0 \mid x :: xs \Rightarrow \widehat{e}_1 : \tau_2}\ \textsc{List-Case}$$

$$\frac{\Gamma \vdash \widehat{e} : \mathsf{real}^m}{\Gamma \vdash -\widehat{e} : \mathsf{real}^m}\ \textsc{Neg} \qquad \frac{\Gamma \vdash \widehat{e}_1 : \mathsf{real}^m \qquad \Gamma \vdash \widehat{e}_2 : \mathsf{real}^m}{\Gamma \vdash \widehat{e}_1 + \widehat{e}_2 : \mathsf{real}^m}\ \textsc{Add}$$

$$\frac{\Gamma \vdash \widehat{e}_1 : \mathsf{real}^{\mathsf{G}} \qquad \Gamma \vdash \widehat{e}_2 : \mathsf{real}^m}{\Gamma \vdash \widehat{e}_1 \cdot \widehat{e}_2 : \mathsf{real}^m}\ \textsc{Mul} \qquad \frac{\Gamma \vdash \widehat{e}_1 : \mathsf{real}^m \qquad \Gamma \vdash \widehat{e}_2 : \mathsf{real}^{\mathsf{G}}}{\Gamma \vdash \widehat{e}_1 / \widehat{e}_2 : \mathsf{real}^m}\ \textsc{Div}$$

$$\frac{\Gamma \vdash \widehat{e}_1 : \mathsf{real}^{\mathsf{G}} \qquad \Gamma \vdash \widehat{e}_2 : \mathsf{real}^{\mathsf{G}}}{\Gamma \vdash \widehat{e}_1 < \widehat{e}_2 : \mathsf{bool}}\ \textsc{Cmp}$$

For the distribution rules below, $m \in \{\mathsf{G}, \mathsf{E}\}$ and $\alpha \in \{m, \mathsf{M}\}$.

$$\frac{\Gamma \vdash \widehat{e}_1 : \mathsf{real}^m \qquad \Gamma \vdash \widehat{e}_2 : \mathsf{real}^m}{\Gamma \vdash \mathsf{uniform}^{\alpha}(\widehat{e}_1, \widehat{e}_2) : \mathsf{real}^m}\ \textsc{Uniform} \qquad \frac{\Gamma \vdash \widehat{e}_1 : \mathsf{real}^m \qquad \Gamma \vdash \widehat{e}_2 : \mathsf{real}^{\mathsf{G}}}{\Gamma \vdash \mathsf{gaussian}^{\alpha}(\widehat{e}_1, \widehat{e}_2) : \mathsf{real}^m}\ \textsc{Gaussian}$$

$$\frac{\Gamma \vdash \widehat{e} : \mathsf{real}^m}{\Gamma \vdash \mathsf{poisson}^{\alpha}(\widehat{e}) : \mathsf{real}^m}\ \textsc{Poisson} \qquad \frac{\Gamma \vdash \widehat{e} : \mathsf{real}^{\mathsf{G}}}{\Gamma \vdash \mathsf{exponential}^{\alpha}(\widehat{e}) : \mathsf{real}^m}\ \textsc{Exponential}$$

$$\frac{\Gamma \vdash \widehat{e} : \mathsf{real}^m}{\Gamma \vdash \mathsf{bernoulli}^{\alpha}(\widehat{e}) : \mathsf{real}^m}\ \textsc{Bernoulli} \qquad \frac{\Gamma \vdash \widehat{e} : \mathsf{list}(\mathsf{real}^m)}{\Gamma \vdash \mathsf{discrete}^{\alpha}(\widehat{e}) : \mathsf{real}^m}\ \textsc{Discrete}$$

$$\frac{\Gamma \vdash \widehat{e}_1 : \mathsf{real}^{\mathsf{G}} \qquad \Gamma \vdash \widehat{e}_2 : \mathsf{real}^{\mathsf{G}}}{\Gamma \vdash \mathsf{beta}^{\alpha}(\widehat{e}_1, \widehat{e}_2) : \mathsf{real}^m}\ \textsc{Beta} \qquad \frac{\Gamma \vdash \widehat{e}_1 : \mathsf{real}^m \qquad \Gamma \vdash \widehat{e}_2 : \mathsf{real}^{\mathsf{G}}}{\Gamma \vdash \mathsf{gamma}^{\alpha}(\widehat{e}_1, \widehat{e}_2) : \mathsf{real}^m}\ \textsc{Gamma}$$

The subtyping relation is unchanged:

$$\frac{}{\tau <: \tau}\ \textsc{Refl} \qquad \frac{\tau <: \sigma \qquad \sigma <: \rho}{\tau <: \rho}\ \textsc{Trans} \qquad \frac{}{\mathsf{real}^{\mathsf{G}} <: \mathsf{real}^{\mathsf{E}}}\ \textsc{Real-Sub}$$

$$\frac{\sigma_1 <: \tau_1 \qquad \tau_2 <: \sigma_2}{\tau_1 \to \tau_2 <: \sigma_1 \to \sigma_2}\ \textsc{Arrow} \qquad \frac{\tau_1 <: \sigma_1 \qquad \tau_2 <: \sigma_2}{\tau_1 \times \tau_2 <: \sigma_1 \times \sigma_2}\ \textsc{Product} \qquad \frac{\tau_1 <: \sigma_1 \qquad \tau_2 <: \sigma_2}{\tau_1 + \tau_2 <: \sigma_1 + \sigma_2}\ \textsc{Sum}$$

$$\frac{\tau <: \sigma}{\mathsf{list}(\tau) <: \mathsf{list}(\sigma)}\ \textsc{List} \qquad \frac{\Gamma \vdash \widehat{e} : \tau \qquad \tau <: \sigma}{\Gamma \vdash \widehat{e} : \sigma}\ \textsc{Sub}$$

J.3 Symbolic States

Definition J.3 (Symbolic state). A symbolic environment and a symbolic state have the forms

$$\sigma ::= \emptyset \mid \sigma,\ u \sim D(\bar{a}),$$
$$s ::= (\sigma \mid \widehat{e}).$$

In a state containing k declarations, every affine expression in $\widehat{e}$ has coordinates $u_1, \ldots, u_k$. A declaration $u_i \sim D_i(\bar{a}_i)$ is well formed when u_i is fresh and its parameters refer only to $u_1, \ldots, u_{i-1}$. The environment is well typed when each declaration is well formed and satisfies $\mathsf{Params}^{\#}_{D_i}(\bar{a}_i)$, defined below. In particular, parameters required to have type $\mathsf{real}^{\mathsf{G}}$ are constant affine expressions. A state is well typed at type τ when its environment is well typed and $\emptyset \vdash \widehat{e} : \tau$. The declarations

provide the coordinates of affine literals; they are not term variables in Γ. When a declaration is appended, existing affine expressions are extended with coefficient zero for the fresh coordinate.

Writing $\sigma_{<i}$ for the declarations preceding u_i, the environment is *domain-safe* if, for every declaration $u_i \sim D_i(\bar{a}_i)$, $\mathrm{Domain}_{D_i}(\bar{a}_i[\rho])$ holds for $\mathcal{A}_{\sigma_{<i}}$-almost every valuation ρ. A state $(\sigma \mid \widehat{e})$ is *domain-safe* if its environment is domain-safe and $\mathrm{realize}_\rho(\widehat{e})$ is domain-safe for $\mathcal{A}_\sigma$-almost every valuation ρ.

J.4 Residual Reduction

Structural residual reduction. The relation $\rightsquigarrow_{\mathrm{sym}}$ reduces a residual redex before it is placed in an evaluation context.

$$
\begin{aligned}
(\lambda x.\widehat{e})\,\widehat{v} &\rightsquigarrow_{\mathrm{sym}} \widehat{e}[x \mapsto \widehat{v}],\\
(\mathsf{rec}\ f\ x.\ \widehat{e})\,\widehat{v} &\rightsquigarrow_{\mathrm{sym}} \widehat{e}[x \mapsto \widehat{v}, f \mapsto \mathsf{rec}\ f\ x.\ \widehat{e}],\\
\mathsf{let}\ x = \widehat{v}\ \mathsf{in}\ \widehat{e} &\rightsquigarrow_{\mathrm{sym}} \widehat{e}[x \mapsto \widehat{v}],\\
\mathsf{reject} &\rightsquigarrow_{\mathrm{sym}} \mathsf{reject},\\
\mathsf{if\ true\ then}\ \widehat{e}_1\ \mathsf{else}\ \widehat{e}_2 &\rightsquigarrow_{\mathrm{sym}} \widehat{e}_1,\\
\mathsf{if\ false\ then}\ \widehat{e}_1\ \mathsf{else}\ \widehat{e}_2 &\rightsquigarrow_{\mathrm{sym}} \widehat{e}_2,\\
\mathsf{fst}\ \langle \widehat{v}_1, \widehat{v}_2 \rangle &\rightsquigarrow_{\mathrm{sym}} \widehat{v}_1,\\
\mathsf{snd}\ \langle \widehat{v}_1, \widehat{v}_2 \rangle &\rightsquigarrow_{\mathrm{sym}} \widehat{v}_2.\\
\mathsf{case}\ \mathsf{inl}\,\widehat{v}\ \mathsf{of}\ \mathsf{inl}\,x \Rightarrow \widehat{e}_1 \mid \mathsf{inr}\,y \Rightarrow \widehat{e}_2 &\rightsquigarrow_{\mathrm{sym}} \widehat{e}_1[x \mapsto \widehat{v}],\\
\mathsf{case}\ \mathsf{inr}\,\widehat{v}\ \mathsf{of}\ \mathsf{inl}\,x \Rightarrow \widehat{e}_1 \mid \mathsf{inr}\,y \Rightarrow \widehat{e}_2 &\rightsquigarrow_{\mathrm{sym}} \widehat{e}_2[y \mapsto \widehat{v}],\\
\mathsf{case}\ []\ \mathsf{of}\ [] \Rightarrow \widehat{e}_0 \mid x :: xs \Rightarrow \widehat{e}_1 &\rightsquigarrow_{\mathrm{sym}} \widehat{e}_0,\\
\mathsf{case}\ \widehat{v} :: \widehat{w}\ \mathsf{of}\ [] \Rightarrow \widehat{e}_0 \mid x :: xs \Rightarrow \widehat{e}_1 &\rightsquigarrow_{\mathrm{sym}} \widehat{e}_1[x \mapsto \widehat{v}, xs \mapsto \widehat{w}].
\end{aligned}
$$

Affine arithmetic and comparisons. Write $a = c_0 + \sum_i c_i u_i$ and $b = d_0 + \sum_i d_i u_i$. Define operations on affine expressions by

$$
\begin{aligned}
\mathrm{neg}(a) &= -c_0 + \sum_i (-c_i) u_i,\\
\mathrm{add}(a,b) &= (c_0 + d_0) + \sum_i (c_i + d_i) u_i,\\
\mathrm{scale}(r,a) &= r c_0 + \sum_i (r c_i) u_i.
\end{aligned}
$$

The corresponding residual reductions are:

$$
\begin{aligned}
-\mathsf{sym}(a) &\rightsquigarrow_{\mathrm{sym}} \mathsf{sym}(\mathrm{neg}(a)),\\
\mathsf{sym}(a) + \mathsf{sym}(b) &\rightsquigarrow_{\mathrm{sym}} \mathsf{sym}(\mathrm{add}(a,b)),\\
\mathsf{sym}(r) \cdot \mathsf{sym}(a) &\rightsquigarrow_{\mathrm{sym}} \mathsf{sym}(\mathrm{scale}(r,a)),\\
\mathsf{sym}(a) \cdot \mathsf{sym}(r) &\rightsquigarrow_{\mathrm{sym}} \mathsf{sym}(\mathrm{scale}(r,a)),\\
\mathsf{sym}(a)/\mathsf{sym}(r) &\rightsquigarrow_{\mathrm{sym}} \mathsf{sym}(\mathrm{scale}(1/r,a)) \qquad (r \neq 0),\\
\mathsf{sym}(r) < \mathsf{sym}(q) &\rightsquigarrow_{\mathrm{sym}} \mathsf{true} \qquad (r < q),\\
\mathsf{sym}(r) < \mathsf{sym}(q) &\rightsquigarrow_{\mathrm{sym}} \mathsf{false} \qquad (r \geq q).
\end{aligned}
$$

Multiplication requires at least one constant operand; division requires a nonzero constant denominator; comparison requires two constant operands. The typing rules ensure the constant-operand

requirements for well-typed residual expressions; domain safety additionally ensures that reached denominators are nonzero almost surely.

Distribution parameters and symbolic means. For E sampling and M evaluation, the underlying affine parameter forms (before componentwise wrapping) are

$$\begin{aligned} \mathsf{uniform}(a,b), &\qquad \mathsf{gaussian}(a,r),\\ \mathsf{poisson}(a), &\qquad \mathsf{exponential}(r),\\ \mathsf{bernoulli}(a), &\qquad \mathsf{discrete}([a_i]_{i<n}),\\ \mathsf{beta}(r,q), &\qquad \mathsf{gamma}(a,r). \end{aligned}$$

Let $\mathrm{Params}^{\#}_D(\bar{a})$ mean that the parameters have the corresponding form above. This is a structural requirement, distinct from the numerical domain condition Domain_D.

The symbolic mean is the following affine expression, where addition and scaling use the operations defined above:

$$\begin{aligned} \mathrm{Mean}^{\#}_{\mathsf{uniform}}(a,b) &= \tfrac{1}{2}(a+b),\\ \mathrm{Mean}^{\#}_{\mathsf{gaussian}}(a,r) &= a,\\ \mathrm{Mean}^{\#}_{\mathsf{poisson}}(a) &= a,\\ \mathrm{Mean}^{\#}_{\mathsf{exponential}}(r) &= 1/r,\\ \mathrm{Mean}^{\#}_{\mathsf{bernoulli}}(a) &= a,\\ \mathrm{Mean}^{\#}_{\mathsf{discrete}}([a_i]_{i<n}) &= n + \sum_{i<n}(i-n)a_i,\\ \mathrm{Mean}^{\#}_{\mathsf{beta}}(r,q) &= r/(r+q),\\ \mathrm{Mean}^{\#}_{\mathsf{gamma}}(a,r) &= (1/r)a. \end{aligned}$$

These formulas are used under the domain-safety hypotheses of the soundness results. Mean evaluation reduces by

$$\frac{\mathrm{Params}^{\#}_D(\bar{a})}{\mathsf{D}^{\mathsf{M}}(\mathsf{sym}(\bar{a})) \rightsquigarrow_{\mathrm{sym}} \mathsf{sym}(\mathrm{Mean}^{\#}_D(\bar{a}))}\ \textsc{Mean}.$$

J.5 Symbolic State Transitions

The arrows distinguish deterministic transitions from sampling actions. Reduce and Sample-E have a single successor, whereas Sample-G specifies a distribution and a continuation mapping each sampled value to its successor state.

$$\frac{\widehat{e}_1 \rightsquigarrow_{\mathrm{sym}} \widehat{e}_2}{(\sigma \mid \widehat{C}[\widehat{e}_1]) \longrightarrow_{\mathrm{sym}} (\sigma \mid \widehat{C}[\widehat{e}_2])}\ \textsc{Reduce}$$

$$\frac{u \notin \mathrm{dom}(\sigma) \qquad \mathrm{Params}^{\#}_D(\bar{a})}{(\sigma \mid \widehat{C}[\mathsf{D}^{\mathsf{E}}(\mathsf{sym}(\bar{a}))]) \longrightarrow_{\mathrm{sym}} (\sigma, u \sim D(\bar{a}) \mid \widehat{C}[\mathsf{sym}(u)])}\ \textsc{Sample-E}$$

$$\frac{\mathrm{Domain}_D(\bar{r})}{(\sigma \mid \widehat{C}[\mathsf{D}^{\mathsf{G}}(\mathsf{sym}(\bar{r}))]) \xrightarrow{D(\bar{r})}_{\mathrm{sym}} \left(r \mapsto (\sigma \mid \widehat{C}[\mathsf{sym}(r)])\right)}\ \textsc{Sample-G}$$

In Sample-G, r is sampled according to $D(\bar{r})$; the continuation specifies the resulting state, not an arbitrary choice of r. All parameters $\bar{r}$ are constant values, including constant list entries for the discrete primitive.

In Sample-E, $\mathsf{sym}(u)$ wraps the affine expression for the newly introduced coordinate. Existing affine expressions are extended with coefficient zero for u. The step records a distribution without evaluating its domain condition at an unknown valuation; domain safety is imposed separately in the soundness statements.

A state whose residual expression is a value is terminal. Apart from the rejection self-loop, expressions for which no rule applies are stuck. Only Sample-G contributes an event to the retained trace.

J.6 Lifting, Realization, and Initialization

Definition J.4 (Lifting [EVN]). Write $L(e) := \mathsf{lift}(e)$. Lifting maps source expressions to residual expressions with no symbolic coordinates. For a real literal,

$$L(r) = \mathsf{sym}(\underline{r}), \qquad \underline{r} = r \quad \text{(the constant affine expression).}$$

Its remaining clauses are:

$$\begin{aligned}
L(x) &= x,\\
L(()) &= (),\\
L(\mathsf{true}) &= \mathsf{true},\\
L(\mathsf{false}) &= \mathsf{false},\\
L(\mathsf{reject}) &= \mathsf{reject},\\
L([]) &= [],\\
L(\lambda x.e) &= \lambda x.L(e),\\
L(\mathsf{rec}\ g\ x.\ e) &= \mathsf{rec}\ g\ x.\ L(e),\\
L(e_1\ e_2) &= L(e_1)\ L(e_2),\\
L(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2) &= \mathsf{let}\ x = L(e_1)\ \mathsf{in}\ L(e_2),\\
L(\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3) &= \mathsf{if}\ L(e_1)\ \mathsf{then}\ L(e_2)\ \mathsf{else}\ L(e_3),\\
L(-e) &= -L(e).
\end{aligned}$$

$$
\begin{aligned}
L\big(\langle e_1, e_2\rangle\big) &= \langle L(e_1), L(e_2)\rangle,\\
L\big(\mathsf{fst}\, e\big) &= \mathsf{fst}\, L(e),\\
L\big(\mathsf{snd}\, e\big) &= \mathsf{snd}\, L(e),\\
L\big(\mathsf{inl}\, e\big) &= \mathsf{inl}\, L(e),\\
L\big(\mathsf{inr}\, e\big) &= \mathsf{inr}\, L(e),\\
L\big(\mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}\, x \Rightarrow e_1 \mid \mathsf{inr}\, y \Rightarrow e_2\big) &= \mathsf{case}\ L(e)\ \mathsf{of}\ \mathsf{inl}\, x \Rightarrow L(e_1) \mid \mathsf{inr}\, y \Rightarrow L(e_2),\\
L\big(e_1 :: e_2\big) &= L(e_1) :: L(e_2),\\
L\big(\mathsf{case}\ e\ \mathsf{of}\ [] \Rightarrow e_0 \mid x :: xs \Rightarrow e_1\big) &= \mathsf{case}\ L(e)\ \mathsf{of}\ [] \Rightarrow L(e_0) \mid x :: xs \Rightarrow L(e_1),\\
L(e_1 + e_2) &= L(e_1) + L(e_2),\\
L(e_1 \cdot e_2) &= L(e_1) \cdot L(e_2),\\
L(e_1 / e_2) &= L(e_1) / L(e_2),\\
L(e_1 < e_2) &= L(e_1) < L(e_2),\\
L\big(D^{\alpha}(e_1, \ldots, e_j)\big) &= D^{\alpha}(L(e_1), \ldots, L(e_j)).
\end{aligned}
$$

Definition J.5 (Realization). For a valuation $\rho : \{u_1, \ldots, u_k\} \to \mathbb{R}$, write $R_\rho(\widehat{e}) := \mathsf{realize}_\rho(\widehat{e})$. Realization evaluates affine literals to ordinary real literals:

$$
R_\rho\left(\mathsf{sym}(c_0 + \sum_{i=1}^{k} c_i u_i)\right) = c_0 + \sum_{i=1}^{k} c_i \rho(u_i).
$$

Its remaining clauses are:

$$
\begin{aligned}
R_\rho\big(x\big) &= x,\\
R_\rho\big(()\big) &= (),\\
R_\rho\big(\mathsf{true}\big) &= \mathsf{true},\\
R_\rho\big(\mathsf{false}\big) &= \mathsf{false},\\
R_\rho\big(\mathsf{reject}\big) &= \mathsf{reject},\\
R_\rho\big([]\big) &= [],\\
R_\rho\big(\lambda x.\widehat{e}\big) &= \lambda x.R_\rho(\widehat{e}),\\
R_\rho\big(\mathsf{rec}\ g\ x.\,\widehat{e}\big) &= \mathsf{rec}\ g\ x.\,R_\rho(\widehat{e}),\\
R_\rho\big(\widehat{e}_1\ \widehat{e}_2\big) &= R_\rho(\widehat{e}_1)\ R_\rho(\widehat{e}_2),\\
R_\rho\big(\mathsf{let}\ x = \widehat{e}_1\ \mathsf{in}\ \widehat{e}_2\big) &= \mathsf{let}\ x = R_\rho(\widehat{e}_1)\ \mathsf{in}\ R_\rho(\widehat{e}_2),\\
R_\rho\big(\mathsf{if}\ \widehat{e}_1\ \mathsf{then}\ \widehat{e}_2\ \mathsf{else}\ \widehat{e}_3\big) &= \mathsf{if}\ R_\rho(\widehat{e}_1)\ \mathsf{then}\ R_\rho(\widehat{e}_2)\ \mathsf{else}\ R_\rho(\widehat{e}_3),\\
R_\rho\big(-\widehat{e}\big) &= -R_\rho(\widehat{e}).
\end{aligned}
$$

$$
\begin{aligned}
R_\rho\big(\langle \widehat{e}_1, \widehat{e}_2 \rangle\big) &= \langle R_\rho(\widehat{e}_1), R_\rho(\widehat{e}_2) \rangle, \\
R_\rho\big(\mathsf{fst}\, \widehat{e}\big) &= \mathsf{fst}\, R_\rho(\widehat{e}), \\
R_\rho\big(\mathsf{snd}\, \widehat{e}\big) &= \mathsf{snd}\, R_\rho(\widehat{e}), \\
R_\rho\big(\mathsf{inl}\, \widehat{e}\big) &= \mathsf{inl}\, R_\rho(\widehat{e}), \\
R_\rho\big(\mathsf{inr}\, \widehat{e}\big) &= \mathsf{inr}\, R_\rho(\widehat{e}), \\
R_\rho\big(\mathsf{case}\ \widehat{e}\ \mathsf{of}\ \mathsf{inl}\, x \Rightarrow \widehat{e}_1 \mid \mathsf{inr}\, y \Rightarrow \widehat{e}_2\big) &= \mathsf{case}\ R_\rho(\widehat{e})\ \mathsf{of}\ \mathsf{inl}\, x \Rightarrow R_\rho(\widehat{e}_1) \mid \mathsf{inr}\, y \Rightarrow R_\rho(\widehat{e}_2), \\
R_\rho\big(\widehat{e}_1 :: \widehat{e}_2\big) &= R_\rho(\widehat{e}_1) :: R_\rho(\widehat{e}_2), \\
R_\rho\big(\mathsf{case}\ \widehat{e}\ \mathsf{of}\ [] \Rightarrow \widehat{e}_0 \mid x :: xs \Rightarrow \widehat{e}_1\big) &= \mathsf{case}\ R_\rho(\widehat{e})\ \mathsf{of}\ [] \Rightarrow R_\rho(\widehat{e}_0) \mid x :: xs \Rightarrow R_\rho(\widehat{e}_1), \\
R_\rho(\widehat{e}_1 + \widehat{e}_2) &= R_\rho(\widehat{e}_1) + R_\rho(\widehat{e}_2), \\
R_\rho(\widehat{e}_1 \cdot \widehat{e}_2) &= R_\rho(\widehat{e}_1) \cdot R_\rho(\widehat{e}_2), \\
R_\rho(\widehat{e}_1 / \widehat{e}_2) &= R_\rho(\widehat{e}_1) / R_\rho(\widehat{e}_2), \\
R_\rho(\widehat{e}_1 < \widehat{e}_2) &= R_\rho(\widehat{e}_1) < R_\rho(\widehat{e}_2), \\
R_\rho\big(D^\alpha(\widehat{e}_1, \ldots, \widehat{e}_j)\big) &= D^\alpha(R_\rho(\widehat{e}_1), \ldots, R_\rho(\widehat{e}_j)).
\end{aligned}
$$

In both definitions, the distribution clause applies to every primitive, with arities

$$
\begin{aligned}
j = 1 \quad & \text{for } D \in \{\text{poisson}, \text{exponential}, \text{bernoulli}, \text{discrete}\}, \\
j = 2 \quad & \text{for } D \in \{\text{uniform}, \text{gaussian}, \text{beta}, \text{gamma}\}.
\end{aligned}
$$

The discrete primitive takes a list expression as its single argument. The mode $\alpha \in \{\mathsf{G}, \mathsf{E}, \mathsf{M}\}$ and all program-variable binders are preserved. Realization does not substitute for program variables.

Definition J.6 (Initialization). For a source expression e, define

$$\mathsf{init}(e) := (\emptyset \mid \mathsf{lift}(e)).$$

With $()$ denoting the empty valuation, $\mathsf{realize}_{()}(\mathsf{lift}(e)) = e$ LEAN.

K Miscellaneous

When Expectations Are Undefined.

Example K.1. In the coin-flipping example (cf. Example 4.3), each terminating G-trace has positive probability. A trace can also contain continuous draws:

```
let u = uniform[G](-1, 1) in
uniform[E](0, 2) / u
```

This program has an undefined global expectation. Yet for each fixed $u \neq 0$, the remaining expression has mean $1/u$, exactly the value returned by the transformed program. Thus determinization is tracewise sound for this program, but the global expectation preservation theorem does not apply.

Moreover, note that every individual value of u has probability zero, so conditioning on a trace cannot simply mean dividing by its probability, hence the disintegration-based definition of conditional output distributions in Equation (1).

Example K.2. Consider

```
let u = uniform[G](0, 1) in
let s = 2 * bernoulli[G](1/2) - 1 in
s / u + uniform[E](-1, 1)
```

The program returns almost surely and is domain-safe: although division is undefined when $u = 0$, this event has probability zero under the continuous uniform distribution. However,

$$\int_0^1 \frac{1}{u}\, du = \infty.$$

When $s = 1$, the term s/u gives the return value an infinite positive part; when $s = -1$, it gives the return value an infinite negative part. Since each sign occurs with probability $1/2$, both parts have infinite expectation, so the program's global expectation is undefined.

Determinization is nevertheless valid and tracewise sound for this program. Determinization replaces the final uniform draw by its mean zero and therefore returns exactly s/u for some fixed G-trace. The global expectation preservation theorem does not apply because its hypothesis requires the source expectation to be well-defined in the extended reals. Likewise, the variance theorem does not apply because the program has an infinite second moment.

Domain safety and missing probability.

Example K.3. The following program is not domain-safe:

```
let x = bernoulli[G](1/2) in
1 / x
```

With probability $1/2$, the sample is $x = 0$, at which point division by zero causes execution to get stuck. Because this invalid operation occurs with positive probability, it cannot be ignored by the almost-sure condition in domain safety. When $x = 1$, the program successfully returns 1, so

$$\mu_e = \frac{1}{2}\delta_1, \qquad \mu_e(\mathbb{R}) = \frac{1}{2}.$$

The missing probability mass comes from an invalid operation rather than rejection or divergence.

Example K.4. The following well-typed program e is not domain-safe:

```
let u = gaussian[E](0, 1) in uniform[E](0, u)
```

When $u < 0$, the uniform distribution's parameters are invalid and execution gets stuck. The program returns with probability $1/2$; conditioned on returning, its expected output is half the mean of a standard Gaussian conditioned to be positive. Determinization sets $u = 0$ and returns the mean of $\mathsf{uniform}(0, 0)$, which is zero. Thus the domain-safety assumption is necessary, because:

$$\mathbb{E}_{\mathrm{ret}}[e] = \frac{1}{\sqrt{2\pi}} \qquad \text{but} \qquad \mathbb{E}_{\mathrm{ret}}[\mathrm{Determinize}(e)] = 0.$$

Why domain safety is necessary. Tracewise soundness and expectation preservation require domain safety (Theorems 4.2 and 4.8). Without domain safety, replacing an E-mode sample by its mean can change whether a later operation is defined, and therefore change which executions contribute to the return-conditioned distribution.

For example,

```
let u = gaussian[E](0, 1) in
uniform[E](0, u)
```

is not domain-safe: executions with $u < 0$ become stuck. Conditioning on successful return therefore implicitly conditions u to be positive, giving

$$\mathbb{E}_{\mathrm{ret}}[e] = \frac{1}{2}\mathbb{E}[u \mid u > 0] = \frac{1}{\sqrt{2\pi}}.$$

Determinization instead replaces u by its unconditional mean 0, yielding

$$\mathbb{E}_{\text{ret}}[\text{Determinize}(e)] = 0.$$

Thus domain safety prevents determinization from changing which executions are semantically valid and, consequently, which executions contribute to the return-conditioned output distribution.

L Full Benchmarks (for RQ1)

L.1 add-noise

```
let addNoise =
  rec addNoise db => fun variance =>
    match db with
    | [] =>
        []
    | y :: ys =>
        let yn = gauss(y, variance) in
        let yns = addNoise ys variance in
        yn :: yns
in

let sum =
  rec sum xs =>
    match xs with
    | [] => 0
    | x :: rest => x + sum rest
in

let database = 1 :: 2 :: 3 :: [] in
let noisy = addNoise database 1 in
sum noisy
```

L.2 add-uniform-counter

```
let equal = fun x => fun y =>
  if x <= y then y <= x else false
in

let f = rec f y =>
  if equal y 2 then
    0
  else
    let left = flip(0.5) in
    let cur =
      if left then
        uniform(0, 0.5)
      else
        uniform(0.5, 1)
    in
```

```
    if left then
      cur + f (y + 1)
    else
      cur + f y
in

f 0
```

L.3 aggregate-rv

```
let aggregate = fun n =>
  let loop = rec loop i => fun x =>
    if i < n then
      let u = uniform(0, 1) in
      loop (i + 1) (x + u)
    else
      x
  in
  loop 0 0
in
aggregate
```

L.4 asym-random-walk

```
let f = rec f x =>
  if flip(0.5) then
    uniform(-1, 1)
  else if 2 <= x then
    uniform(-1, 0) + f 0
  else
    uniform(0, 1) + f (x + 1)
in

f 0
```

L.5 dreckon

```
let n = 2 in
let loop =
  rec loop i => fun x => fun y => fun estX => fun estY =>
    if i <= n then
      let cmd = discrete(0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.2) in
      let displacement =
        if cmd < 1 then
          (0, uniform(1, 2))
        else if cmd < 2 then
          (0, -uniform(1, 2))
        else if cmd < 3 then
          (uniform(1, 2), 0)
```

```
            else if cmd < 4 then
              (-uniform(1, 2), 0)
            else if cmd < 5 then
              (uniform(1, 2), uniform(1, 2))
            else if cmd < 6 then
              (uniform(1, 2), -uniform(1, 2))
            else if cmd < 7 then
              (-uniform(1, 2), uniform(1, 2))
            else if cmd < 8 then
              (-uniform(1, 2), -uniform(1, 2))
            else
              (0, 0)
          in
          let dxc = fst displacement in
          let dyc = snd displacement in
          let dx = dxc + uniform(-0.05, 0.05) in
          let dy = dyc + uniform(-0.05, 0.05) in
          loop (i + 1) (x + dx) (y + dy) (estX + dxc) (estY + dyc)
        else
          x
    in
    loop 0 0 0 0 0
```

L.6 gauss-random-walk

```
    let walk =
      rec walk times => fun time => fun position =>
        match times with
        | [] =>
            position
        | nextTime :: remainingTimes =>
            let nextPosition =
              gauss(position, nextTime - time)
            in
            walk remainingTimes nextTime nextPosition
    in

    let times = 1 :: 2 :: 3 :: 4 :: 5 :: 6 :: [] in
    walk times 0 0
```

L.7 invpend

```
    let invpend = fun n =>
      let cartPos = uniform(-5, -3) in
      let cartVelo = uniform(2, 3) in
      let pAng = uniform(1, 5) in
      let pAngDer = uniform(-0.5, 0.5) in
```

```
    let loop = rec loop count => fun cartPos => fun cartVelo => fun pAng => fun pAngDer
        =>
      if count < n then
        let count = count + 1 in
        let cartPos = cartPos + 0.01 * cartVelo in
        let cartVelo =
          0.02 * cartPos
          + 1.03 * cartVelo
          - 0.3 * pAng
          - 0.06 * pAngDer
          + uniform(-1, 1)
        in
        let pAng = pAng + 0.01 * pAngDer in
        let pAngDer =
          0.04 * cartPos
          + 0.07 * cartVelo
          - 0.51 * pAng
          + 0.85 * pAngDer
          + uniform(-0.8, 0.8)
        in
        loop count cartPos cartVelo pAng pAngDer
      else
        pAng
    in
    loop 0 cartPos cartVelo pAng pAngDer
  in
  invpend
```

L.8 monte-carlo

```
  let monteCarlo =
    fun draw => fun h =>
      rec loop i =>
        if i <= 0 then
          0
        else
          let m = loop (i - 1) in
          let x = draw () in
          (h x + (i - 1) * m) / i
  in

  let draw = fun unit => uniform(0, 1) in
  let h = fun x => 2 * x + 1 in

  monteCarlo draw h 10
```

L.9 noisy-iteration

```
(rec run x =>
  if flip(0.5) then x
  else run (1 - x + gaussian(0, 1))
) 0
```

L.10 pack

```
let pack = fun n =>
  let loop =
    rec loop nLight => fun nHeavy => fun nPacked =>
      if nPacked < n then
        let obj = discrete(0.5, 0.25, 0.25) in

        if obj < 1 then
          let weight = 0.9 + uniform(0, 0.2) in
          weight + loop nLight nHeavy (nPacked + 1)

        else if obj < 2 then
          let weight = 1.1 + uniform(0, 0.1) in
          if nHeavy <= nLight then
            weight + loop nLight (nHeavy + 1) (nPacked + 1)
          else
            weight + loop nLight nHeavy nPacked

        else
          let weight = 0.8 + uniform(0, 0.1) in
          if nLight <= nHeavy then
            weight + loop (nLight + 1) nHeavy (nPacked + 1)
          else
            weight + loop nLight nHeavy nPacked
      else
        0
  in
  loop 0 0 0
in
pack
```

L.11 retransmit

```
let receive = rec receive good =>
  let capacity =
    if good then
      uniform(8, 12)
    else
      uniform(2, 6)
  in

  let quality = uniform(0, capacity) in
```

```
    let success =
      if good then
        flip(0.85)
      else
        flip(0.30)
    in

    if success then
      quality
    else
      let next =
        if good then flip(0.80)
        else flip(0.25)
      in
      quality + receive next
  in

  receive true
```

L.12 service-policy

```
  let service_policy = fun n =>
    let reuse = fun warm =>
      let next =
        if warm then flip(0.9)
        else flip(0.4)
      in

      let latency =
        if next then exponential(2.0)
        else exponential(1.0)
      in

      (next, latency)
    in

    let refresh = fun warm =>
      let next = flip(0.95) in

      let latency =
        if next then exponential(1.5)
        else exponential(0.75)
      in

      (next, latency)
    in
```

```
    let run = rec run step => fun n => fun warm =>
      if n < 1 then
        0
      else
        let result = step warm in
        let next = fst result in
        let latency = snd result in
        latency + run step (n - 1) next
    in

    let policy =
      if flip(0.5) then reuse else refresh
    in

    run policy n false
  in
  service_policy
```

L.13 swap

```
  let swap = fun n =>
    let x = 5 * bernoulli(0.5) in
    let y = 4 + 3 * bernoulli(0.5) in
    let loop = rec loop i => fun x => fun y =>
      if i < n then
        let unif1 = uniform(-1, 1) in
        let unif2 = uniform(-2, 2) in
        loop (i + 1) (y + unif1) (x + unif2)
      else
        x
    in
    loop 0 x y
  in
  swap
```

L.14 uniform-nested

```
  let a = uniform(0,1) in
  let b = uniform(0,a) in
  let c = uniform(0,b) in
  let d = uniform(0,c) in d
```

L.15 workload

```
  let workload = fun n =>
    let make_jobs = rec make_jobs n =>
      if n < 1 then
        []
      else
```

```
        let major = flip(0.25) in
        major :: make_jobs (n - 1)
    in

    let repair = rec repair jobs =>
      match jobs with
      | [] =>
          0

      | major :: rest =>
          let time =
            if major then uniform(1.5, 2.5)
            else uniform(0.5, 1.5)
          in

          time + repair rest
    in

    let jobs = make_jobs n in
    repair jobs
  in
  workload
```

M Determinization Scaling

The script `benchmarks/exact/scaling/run_scaling.py` generates three families of programs indexed by a size $k \in \{10, 25, 50, 100, 200, 400\}$:

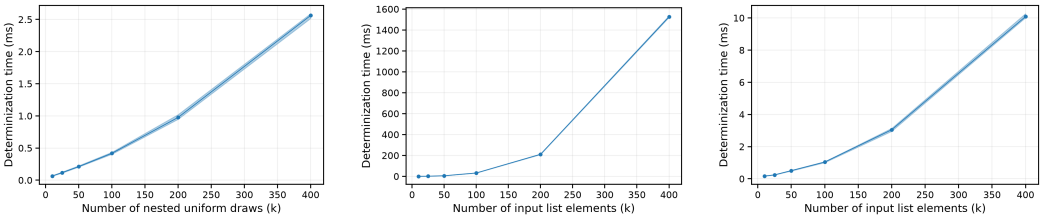
- *Uniform nested* (Fig. 11a): the benchmark `uniform-nested` of Table 1, with its four nested draws extended to k: $x_0 = 1$ and $x_i = \mathsf{uniform}(0, x_{i-1})$ for $i = 1, \ldots, k$, returning x_k. It has k continuous sampling sites, each depending on the previous one, and determinization replaces all of them.
- *Gaussian random walk* (Fig. 11b): the benchmark `gauss-random-walk` of Table 1, with the list of requested times replaced by $[1, \ldots, k]$.
- *Add noise* (Fig. 11c): the benchmark `add-noise` of Table 1, with its input list replaced by $[1, \ldots, k]$.

In the last two families, k is the length of an explicit input list, and the program has one continuous sampling site, which determinization replaces. Figure 11 shows the median and the range between the 25th and 75th percentiles, across 10 runs.

N Full Benchmarks (for RQ2)

For `2d-walk`, `bimodal`, `random-walk-1d`, and `vasicek`, we replace fixed iteration bounds with a `flip(1/2)` continuation guard. The number of iterations is geometrically distributed, with mean one and no finite upper bound. These loops terminate almost surely and retain infinitely many reachable states after determinization. Their stopping coins remain random; in particular, `vasicek` is partially determinized and retains variance from its stopping time. The `hmm` benchmark retains its fixed sequence of eight observations and becomes finite-state after determinization.

For `cont-sampling`, we draw the downward-step probability from $\mathsf{uniform}(1/3, 1)$, so the walk moves toward zero with probability $2/3$ at each step. Starting from 10, it terminates almost surely



(a) `uniform-nested` (b) `gauss-random-walk` (c) `add-noise`

Fig. 11. Determinization time versus the size parameter k. Points show median times over 10 runs; shaded bands show the interquartile range. Timings include determinization only.

after 30 steps on average. This bias gives a finite source expectation of 5 and variance $5/2$; determinization returns 5 on every terminating run. The unbiased walk has an undefined expectation and infinite second moment, so it cannot serve as a finite-moment variance benchmark.

N.1 2d-walk

```
let loop = rec loop x => fun y => fun distance =>
  if flip(1/2) then
    let direction = discrete(0.25, 0.25, 0.25, 0.25) in
    let step = uniform(0, 10) in
    if direction < 1 then
      loop (x + step) y distance
    else if direction < 2 then
      loop (x - step) y distance
    else if direction < 3 then
      loop x (y + step) distance
    else
      loop x (y - step) distance
  else
    x
in
loop 0 0 0
```

N.2 bimodal

```
let loop = rec loop x => fun xlow => fun xup =>
  if flip(1/2) then
    let d = bernoulli(1/3) in
    if d < 1 then
      let step = gauss(3/4, 4) in
      let xlow = xlow - step in
      loop xlow xlow xup
    else
      let step = gauss(3/4, 4) in
      let xup = xup + step in
      loop xup xlow xup
```

```
    else
      x
in
loop 0 0 0
```

N.3 branching

```
let offspring_lambda = 0.2 in
let rate = 1.0 in

let branch = rec branch unit =>
  let k = poisson(offspring_lambda) in
  let lifetime = exponential(rate) in

  let sum_children = rec sum_children remaining => fun total =>
    if remaining < 1 then total
    else
      let child = branch () in
      sum_children (remaining - 1) (total + child)
  in

  let children = sum_children k 0 in
  lifetime + children
in

branch ()
```

N.4 cont-sampling

```
let run = fun n => fun b =>
  let loop = rec loop n => fun b =>
    if 0 < n then
      let p = uniform(1/3, 1) in
      if flip(p) then
        loop (n - 1) (b + uniform(0, 1))
      else
        loop (n + 1) (b - uniform(0, 1))
    else
      b
  in
  loop n b
in
run 10 0
```

N.5 ckd-epi

```
let logScr = uniform[G](-0.6, 0.6) in
let age = uniform(30, 80) in
let isFemale = uniform[G](0, 1) in
```

```
  let isAA = uniform[G](0, 0.55) in
  let f = 4.94 in
  let f =
    if 0.5 <= isFemale then
      let k = -0.357 in
      (if logScr <= k then
         f - 0.329 * (logScr - k)
       else
         f - 1.209 * (logScr - k))
      + 0.017
    else
      let k = -0.105 in
      if logScr <= k then
        f - 0.411 * (logScr - k)
      else
        f - 1.209 * (logScr - k)
  in
  let f = if 0.5 <= isAA then f + 0.148 else f in
  let logScrErr = logScr + uniform[G](-0.1, 0.1) in
  let ageErr = age + uniform(-1, 1) in
  let flip1 = uniform[G](0, 1) in
  let isFemaleErr =
    if flip1 <= 0.01 then 1 - isFemale else isFemale
  in
  let flip2 = uniform[G](0, 1) in
  let isAAErr =
    if flip2 <= 0.01 then 1 - isAA else isAA
  in
  let f1 = 4.94 in
  let f1 =
    if 0.5 <= isFemaleErr then
      let k = -0.357 in
      (if logScrErr <= k then
         f1 - 0.329 * (logScrErr - k)
       else
         f1 - 1.209 * (logScrErr - k))
      + 0.017
    else
      let k = -0.105 in
      if logScr <= k then
        f1 - 0.411 * (logScrErr - k)
      else
        f1 - 1.209 * (logScrErr - k)
  in
  let f1 = if 0.5 <= isAAErr then f1 + 0.148 else f1 in
  if f1 <= 4.4 then
    if 4.6 <= f then 0 else 1
  else

```

```
  1
```

N.6 dungeon

```
let crawl = fun loot =>
  (rec visit depth =>
    if flip(0.25) then 0
    else loot depth + visit (depth + 1)
  ) 1
in
crawl (fun depth => 200 * depth * bernoulli(0.01))
```

N.7 figure1

```
let run = fun s0 =>
  let steps = 1 in
  let loop = rec loop i => fun toggle => fun sum => fun x => fun y => fun z =>
    if i < steps then
      let toggle = 1 - toggle in
      let state =
        if toggle < 1 then
          let x = if flip(1/2) then x + 1 else x + 2 in
          let y = if flip(1/3) then y + z + x * x else y - z - x in
          let z = if flip(1/4) then z + y else z - y in
          (x, (y, z))
        else
          (x, (y, z))
      in
      let x = fst state in
      let y = fst (snd state) in
      let z = snd (snd state) in
      let l = x + y + exponential(1) - exponential(1) in
      let g = gauss[G](0, 1) in
      let sum = if g < 1/2 then sum + x else sum in
      loop (i + 1) toggle sum x y z
    else
      sum
  in
  loop 0 0 s0 1 1 1
in
run 0
```

N.8 growing-walk

```
let f = rec f x =>
  if flip(0.5) then
    0
  else
    x * uniform(0, 1) + f (x + 1)
```

```
in

let t = f 1 in
t
```

N.9 hmm

```
let index = rec index xs => fun i =>
  match xs with
  [] => 1 / 0
  | x :: rest =>
      if i < 1 then x else index rest (i - 1)
in

let equal = fun x => fun y =>
  if x <= y then y <= x else false
in

let emit = 0.05 :: 0.25 :: [] in
let dwell_rate = 0.8 :: 1.2 :: [] in
let trans = 0.05 :: 0.5 :: [] in
let initial_state = 0 in
let data = 0 :: 0 :: 0 :: 0 :: 0 :: 0 :: 0 :: 0 :: [] in

let hmm = rec hmm z => fun data =>
  match data with
  [] => 0
  | data :: remaining_data =>

      let y = bernoulli(index emit z) in
      let _ = observe(equal y data) in

      let dwell = exponential(index dwell_rate z) in

      let next_z = bernoulli(index trans z) in

      let rest = hmm next_z remaining_data in

      dwell + rest
in

hmm initial_state data
```

N.10 load

```
let local = fun load =>
  let cpu =
    uniform(
```

```
        1.0 + 0.2 * load * load,
        3.0 + 0.2 * load * load
      )
  in
  let transfer = exponential(1.0) in
  cpu + transfer
in

let remote = fun load =>
  let cpu =
    uniform(
      0.8 + 0.2 * load * load,
      2.8 + 0.2 * load * load
    )
  in
  let transfer = exponential(0.9) in
  cpu + transfer
in

let run = rec run worker => fun n => fun load =>
  if n < 1 then
    0
  else
    worker load + run worker (n - 1) load
in

let load = uniform(0, 1) in

let worker =
  if load < 0.6 then
    local
  else
    remote
in

run worker 40 load
```

N.11 neals-funnel

```
let x = gauss(0, 3) in

let z = x / 2 in
let z2 = z * z in
let exp_approx =
  (1 + z / 2 + z2 / 12) /
  (1 - z / 2 + z2 / 12)
in
```

```
  let y0 = gauss(0, exp_approx) in
  let y1 = gauss(0, exp_approx) in
  let y2 = gauss(0, exp_approx) in
  let y3 = gauss(0, exp_approx) in
  let y4 = gauss(0, exp_approx) in
  let y5 = gauss(0, exp_approx) in
  let y6 = gauss(0, exp_approx) in
  let y7 = gauss(0, exp_approx) in
  let y8 = gauss(0, exp_approx) in
  let y9 = gauss(0, exp_approx) in

  y0
```

N.12 noisy-product

```
  let x = uniform(0, 1) in
  let y = gaussian(x, 1) in
  x * y
```

N.13 random-walk-1d

```
  let v = 0 in
  let x = 0 in
  let loop = rec loop v => fun x =>
    if flip(1/2) then
      let v = uniform(0, 1) in
      let x =
        if flip(0.9) then
          x + v
        else
          x - v
      in
      loop v x
    else
      x
  in

  loop v x
```

N.14 vasicek

```
  let a = 0.5 in
  let b = 0.2 in
  let sigma = 0.8 in
  let w = 0 in
  let r = 2 in
  let loop = rec loop w => fun r =>
    if flip(1/2) then
      let w = gauss(0, 1) in
```

```
      let r = (1 - a) * r + a * b + sigma * w in
      loop w r
    else
      r
  in

  loop w r
```

Benchmark	Det.	Static sampling sites (source → det.)	Source $\hat{\mu}(P)$	Source $\widehat{\mathrm{Var}}(P)$	Determinized $\hat{\mu}(\det P)$	Determinized $\widehat{\mathrm{Var}}(\det P)$	VRF
2d-walk [9]	•∘	$(2,1) \to (2,0)$	0.00	16.61	0.01	12.63	1.32×
bimodal [45]	•∘	$(2,2) \to (2,0)$	−0.25	3.95	−0.25	0.88	4.51×
branching	•∘	$(1,1) \to (1,0)$	1.25	1.64	1.25	0.39	4.18×
cont-sampling [26]	•∘	$(1,3) \to (1,1)$	5.00	2.53	5.00	0.00	∞
ckd-epi [11]	•∘	$(0,8) \to (0,6)$	1.00	2.80e-4	1.00	2.80e-4	1.00×
dungeon	•∘	$(2,0) \to (1,0)$	23.79	38120.34	24.10	2928.50	13.02×
figure1 [45]	•∘	$(3,3) \to (3,1)$	0.69	0.21	0.69	0.21	1.00×
growing-walk [11]	•∘	$(1,1) \to (1,0)$	1.01	6.47	1.01	5.91	1.09×
hmm	•∘	$(2,1) \to (2,0)$	9.81	12.27	9.85	0.11	108.13×
load	•∘	$(0,5) \to (0,1)$	121.24	57.75	121.24	1.43	40.47×
neals-funnel [11]	•∘	$(0,11) \to (0,1)$	−0.01	1.46	0.00	0.00	∞
noisy-product	•∘	$(0,2) \to (0,1)$	0.33	0.42	0.33	0.09	4.73×
random-walk-1d [8, 45]	•∘	$(2,1) \to (2,0)$	0.40	0.49	0.40	0.41	1.20×
vasicek [34]	•∘	$(1,1) \to (1,0)$	1.39	0.78	1.39	0.41	1.90×

Table 2. Evaluation of determinization on 14 benchmarks, where P is the source program and $\det P$ its determinization. All benchmarks are partially determinized (•∘). Each pair (d, c) counts static discrete and continuous sampling sites. We run 50,000 executions per program; hmm returns in 29,904 runs and rejects the rest, while every run of the other benchmarks returns. No execution reaches the step limit or fails. The variance reduction factor, shown in the last column, is calculated as $\mathrm{VRF} = \widehat{\mathrm{Var}}(P)/\widehat{\mathrm{Var}}(\det P)$, and can be interpreted as the determinized program needing approximately that many times fewer independent samples to obtain the same variance.

O Lean Declarations

Each Lean logo in this paper links to its entry below, which names the Lean declarations that state the marked item and the lines that hold them in the supplementary material. Paths are relative to the root of the archive. The website in it shows each file with line numbers under `website/source/`, and the API documentation of the formalization under `website/docs/`.

Language grammar. `Determinize.Spec.Paper.Expr` in `lean/Determinize/Spec/Syntax.lean`, lines 22–49.

Values. `Determinize.Spec.Paper.Expr.isValue` in `lean/Determinize/Spec/Syntax.lean`, lines 55–59.

Evaluation contexts. `Determinize.Spec.Paper.Action.wrap` in `lean/Determinize/Spec/Semantics.lean`, lines 23–28.

Deterministic reduction. `Determinize.Spec.Paper.reduce` in `lean/Determinize/Spec/Semantics.lean`, lines 45–183.

Primitive distributions. `Determinize.Spec.Paper.uniformFiber`, `Determinize.Spec.Paper.gaussianFiber`, `Determinize.Spec.Paper.poissonFiber`, `Determinize.Spec.Paper.exponentialFiber`, `Determinize.Spec.Paper.betaFiber`, `Determinize.Spec.Paper.gammaFiber`, `Determinize.Spec.Paper.bernoulliFiber`, `Determinize.Spec.Paper.discreteFiber` in `lean/Determinize/Spec/Primitives.lean`, lines 37–115.

Output measures. `Determinize.Spec.Paper.outputMeasureAt`, `Determinize.Spec.Paper.bigStepMeasure` in `lean/Determinize/Spec/Semantics.lean`, lines 186–199.

Determinization. `Determinize.Spec.Paper.Expr.determinize` in `lean/Determinize/Spec/Syntax.lean`, lines 122–162.

Domain safety. `Determinize.Spec.Paper.DomainSafe` in `lean/Determinize/Spec/Semantics.lean`, lines 217–232.

Type grammar. `Determinize.Spec.Paper.Ty` in `lean/Determinize/Spec/Types.lean`, lines 25–31.

Real typing. `Determinize.Spec.Paper.Typed` in `lean/Determinize/Spec/Syntax.lean`, lines 212–286.

Subtyping. `Determinize.Spec.Paper.Ty.Sub` in `lean/Determinize/Spec/Types.lean`, lines 34–42.

Determinization preserves typing. `Determinize.Theorems.determinization_preserves_typing` in `lean/Determinize/Theorems.lean`, lines 54–57.

Constraint generation. `Determinize.Frontend.generate` in `lean/Determinize/Frontend/Infer.lean`, lines 150–231.

Type shape unification. `Determinize.Frontend.unify` in `lean/Determinize/Frontend/Unify.lean`, lines 149–174.

Affinity decoration of type shapes. `Determinize.Frontend.Shape.decorate`, `Determinize.Frontend.UType.decorate` in `lean/Determinize/Frontend/Infer.lean`, lines 50–82.

Subtype constraint reduction. `Determinize.Frontend.decompose` in `lean/Determinize/Frontend/Infer.lean`, lines 104–110.

Mode constraint solving. `Determinize.Frontend.solveAffinities` in `lean/Determinize/Frontend/Affinity.lean`, lines 50–81.

Inference correctness. `Determinize.Theorems.inference_correctness` in `lean/Determinize/Theorems.lean`, lines 41–50.

Expectation. `Determinize.Spec.HasExpectation`, `Determinize.Spec.extendedExpectation` in `lean/Determinize/Spec/Expectation.lean`, lines 21–34.

Return probability. `Determinize.Spec.returnProbability` in `lean/Determinize/Spec/Expectation.lean`, lines 36–37.

Expectation conditioned on returning. `Determinize.Spec.returnedExpectation` in `lean/Determinize/Spec/Expectation.lean`, lines 44–48.

Variance conditioned on returning. `Determinize.Spec.returnedVariance` in `lean/Determinize/Spec/Expectation.lean`, lines 39–52.

Expectation preservation. `Determinize.Theorems.expectation_preservation` in `lean/Determinize/Theorems.lean`, lines 114–124.

Variance non-increase. `Determinize.Theorems.variance_non_increase` in `lean/Determinize/Theorems.lean`, lines 126–136.

Traces. `Determinize.Spec.Traces.Trace` in
`lean/Determinize/Spec/Traces/Semantics.lean`, lines 18–27.
Refined output measures. `Determinize.Spec.Traces.traceAndOutputLaw` in
`lean/Determinize/Spec/Traces/Semantics.lean`, lines 29–51.
Trace marginal. `Determinize.Spec.Traces.traceLaw` in
`lean/Determinize/Spec/Traces/Semantics.lean`, lines 53–55.
Output marginal. `Determinize.Theorems.trace_erasure` in
`lean/Determinize/Theorems.lean`, lines 61–65.
Equality of total masses. `Determinize.Theorems.equality_of_total_masses` in
`lean/Determinize/Theorems.lean`, lines 67–72.
Trace preservation. `Determinize.Theorems.trace_preservation` in
`lean/Determinize/Theorems.lean`, lines 74–79.
Tracewise soundness. `Determinize.Theorems.tracewise_soundness` in
`lean/Determinize/Theorems.lean`, lines 81–91.
Expectation along traces. `Determinize.Theorems.returned_expectation_along_traces` in
`lean/Determinize/Theorems.lean`, lines 93–104.
Output-mass preservation. `Determinize.Theorems.output_mass_preservation` in
`lean/Determinize/Theorems.lean`, lines 108–112.
Divergence. `Determinize.Spec.Paper.divergenceProbability` in
`lean/Determinize/Spec/Semantics.lean`, lines 201–215.
Probabilistic type safety. `Determinize.Theorems.probabilistic_type_safety` in
`lean/Determinize/Theorems.lean`, lines 147–156.
Return or divergence. `Determinize.Theorems.return_or_diverge` in
`lean/Determinize/Theorems.lean`, lines 176–181.
Residual symbolic expressions. `Determinize.Proof.Paper.Symbolic.AffineExpr` in
`lean/Determinize/Proof/Symbolic/Syntax.lean`, line 21.
Affine expressions. `Determinize.Proof.Paper.Symbolic.Affine` in
`lean/Determinize/Proof/Semantics/Internal.lean`, lines 60–63.
Symbolic environments. `Determinize.Proof.Paper.Symbolic.SampleEnv` in
`lean/Determinize/Proof/Semantics/Internal.lean`, lines 119–123.
Actual valuation. `Determinize.Proof.Paper.Symbolic.SampleEnv.actualMeasure` in
`lean/Determinize/Proof/Semantics/Internal.lean`, lines 127–134.
Symbolic reduction. `Determinize.Proof.Paper.Symbolic.AffineExpr.symbolicReduce` in
`lean/Determinize/Proof/Symbolic/Syntax.lean`, lines 1285–1528.
Symbolic reduction actions.
`Determinize.Proof.Paper.Symbolic.AffineExpr.SymbolicAction` in
`lean/Determinize/Proof/Symbolic/Syntax.lean`, lines 1134–1143.
Sequential mean valuation.
`Determinize.Proof.Paper.Symbolic.SampleEnv.meanEnvironment` in
`lean/Determinize/Proof/Semantics/Internal.lean`, lines 145–151.
Lifting. `Determinize.Proof.Paper.Symbolic.AffineExpr.ofExpr` in
`lean/Determinize/Proof/Symbolic/Syntax.lean`, lines 116–151.
Realization. `Determinize.Proof.Paper.Symbolic.AffineExpr.realize` in
`lean/Determinize/Proof/Symbolic/Syntax.lean`, lines 53–102.
Realization of initialization.
`Determinize.Proof.Paper.Symbolic.AffineExpr.realize_ofExpr` in
`lean/Determinize/Proof/Symbolic/Syntax.lean`, lines 827–839.

Actual interpretation. `Determinize.Proof.StepTraces.actualTraceLaw` in `lean/Determinize/Proof/Symbolic/TraceLaws.lean`, lines 49–51.
Expected interpretation. `Determinize.Proof.StepTraces.targetTraceLaw` in `lean/Determinize/Proof/Symbolic/TraceLaws.lean`, lines 55–57.
Expectation of an affine expression. `Determinize.Proof.Paper.SymbolicSoundness.SampleEnv.integral_affine` in `lean/Determinize/Proof/Symbolic/Moments.lean`, lines 99–104.
Trace-indexed output kernel. `Determinize.Proof.StepTraces.compactHistoryReplay`, `Determinize.Proof.StepTraces.compactFiber` in `lean/Determinize/Proof/Traces/CompactFiberSoundness.lean`, lines 23–136.
Finite-step factorization. `Determinize.Proof.StepTraces.compact_exactDepth_fiberSound` in `lean/Determinize/Proof/Traces/CompactFiberSoundness.lean`, lines 193–196.
Whole-program output kernel. `Determinize.Proof.Traces.normalizedKernel` in `lean/Determinize/Proof/Traces/CompactSoundness.lean`, lines 42–44.
Replay compatibility. `Determinize.Proof.StepTraces.outputGivenTrace_eq_outputGivenTraceAt` in `lean/Determinize/Proof/Traces/CompactReplay.lean`, lines 419–428.
Combining finite-step factorizations. `Determinize.Proof.Traces.joint_normalized_fiberSound` in `lean/Determinize/Proof/Traces/CompactSoundness.lean`, lines 77–98.
Whole-program factorization. `Determinize.Proof.Traces.compact_normalized_fiberSound` in `lean/Determinize/Proof/Traces/CompactSoundness.lean`, lines 101–105.
Parsing and elaboration. `Determinize.Frontend.compile` in `lean/Determinize/Frontend/Compile.lean`, lines 20–26.
Floating-point interpreter. `Determinize.Runtime.runOutcome` in `lean/Determinize/Runtime/Eval.lean`, lines 135–141.
Finite graph construction. `Determinize.Finite.Reward.explore` in `lean/Determinize/Finite/Reward/Explore.lean`, lines 13–69.
Exact rational solver. `Determinize.Finite.Reward.solve` in `lean/Determinize/Finite/Reward/Solve.lean`, lines 80–92.
Certificate checker. `Determinize.Finite.Reward.Solution` in `lean/Determinize/Finite/Reward/Solve.lean`, lines 48–62.