



# Formally Verified Samplers from Probabilistic Programs with Loops and Conditioning



ALEXANDER BAGNALL, Ohio University, USA
GORDON STEWART, BedRock Systems, Inc., USA
ANINDYA BANERJEE, IMDEA Software Institute, Spain

We present Zar: a formally verified compiler pipeline from discrete probabilistic programs with unbounded loops in the conditional probabilistic guarded command language (cpGCL) to proved-correct executable samplers in the random bit model. We exploit the key idea that all discrete probability distributions can be reduced to unbiased coin-flipping schemes. The compiler pipeline first translates cpGCL programs into *choice-fix* trees, an intermediate representation suitable for reduction of biased probabilistic choices. Choice-fix trees are then translated to coinductive interaction trees for execution within the random bit model. The correctness of the composed translations establishes the *sampling equidistribution theorem*: compiled samplers are correct wrt. the conditional weakest pre-expectation semantics of cpGCL source programs. Zar is implemented and fully verified in the Coq proof assistant. We extract verified samplers to OCaml and Python and empirically validate them on a number of illustrative examples.

CCS Concepts: • **Software and its engineering** → **Software verification**; • **Theory of computation** → **Probabilistic computation**.

Additional Key Words and Phrases: Probabilistic Programming, Verified Compilers

**ACM Reference Format:**
Alexander Bagnall, Gordon Stewart, and Anindya Banerjee. 2023. Formally Verified Samplers from Probabilistic Programs with Loops and Conditioning. *Proc. ACM Program. Lang.* 7, PLDI, Article 106 (June 2023), 24 pages. https://doi.org/10.1145/3591220

## 1 INTRODUCTION

Probabilistic programming languages [Bingham et al. 2019; Goodman et al. 2012; Gordon et al. 2014] formalize probabilistic systems by modeling them as programs with random sampling and conditioning. Unlike conventional programs, for which meaning is deduced from executions over states or sets of states, probabilistic programs are defined by their posterior distributions for given inputs. Calculating this posterior distribution is called *inference*. In cases in which it is infeasible to calculate the posterior directly, probabilistic programming languages (PPLs) typically support sampling from this distribution. Many standard semantic notions such as weakest precondition transformers have analogues – e.g., weakest pre-expectation transformers – in PPLs.

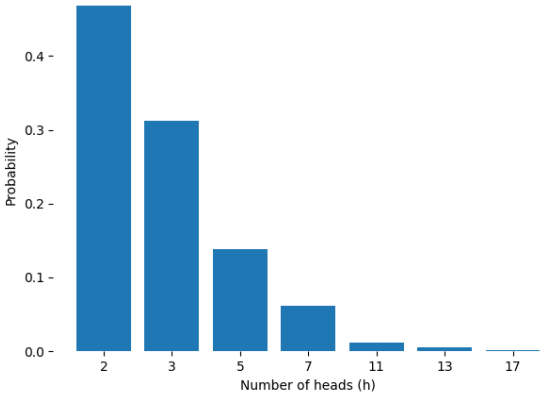
Inference on probabilistic programs (PPs) is automated by compiling the programs to Markov Chain Monte Carlo (MCMC) samplers [Huang et al. 2017] or to other specialized representations [Holtzen et al. 2020, 2019]. Similarly, systems like Pyro [Bingham et al. 2019] use techniques for semi-automated inference. Automation helps separate concerns: the programmer specifies a

Authors' addresses: Alexander Bagnall, abagnalla@gmail.com, Ohio University, Athens, Ohio, USA; Gordon Stewart, BedRock Systems, Inc., Boston, Massachusetts, USA, gordon@bedrocksystems.com; Anindya Banerjee, IMDEA Software Institute, Pozuelo de Alarcon, Madrid, Spain, anindya.banerjee@imdea.org.


2475-1421/2023/6-ART106
https://doi.org/10.1145/3591220

```
1  primes (p : ℚ) :=
2    { b ← true } [p] { b ← false };
3    while b do
4      h ← h + 1;
5      { b ← true } [p] { b ← false }
6    end;
7    observe h is prime
```

(a) 'primes' cpGCL program with geometric posterior over the prime numbers.




(b) True posterior over $h$ with $p = \frac{2}{3}$.

Fig. 1. Geometric primes program (left) and its posterior distribution over $h$ (right).

probabilistic model in a convenient high-level language, and the inference engine takes care of the details of calculating the posterior distribution. At the same time, errors in execution models of probabilistic systems are especially difficult to detect and diagnose [Dutta et al. 2018, 2019], and attempts at empirical validation may fail to detect small biases and low-probability error conditions. The standard belief propagation algorithm for inference may converge to the wrong equilibrium or fail to converge at all [Yedidia et al. 2003], and MCMC samplers may falsely appear to have converged to the desired stationary distribution (known as "pseudo-convergence") [Geyer 2011]. Even the straightforward task of uniform sampling is notoriously susceptible to "modulo bias" [Security 2020], leading to violations of cryptographic guarantees [Aranha et al. 2020] due to improper use of the modulus operator to restrict the range of the uniform distribution.

We implement Zar: a formally verified compiler from the conditional probabilistic guarded command language (cpGCL [Olmedo et al. 2018]) to proved-correct samplers in the random bit model [Saad et al. 2020b], in which samplers are provided a stream of independent and identically distributed (i.i.d.) random bits drawn from a *uniform* distribution. Samplers generated by Zar are guaranteed, under reasonable assumptions about the source of randomness (Section 4.3), to produce samples from the true posterior of their source programs, and thus provide a foundation for high-assurance sampling and Monte Carlo-based [Rubinstein and Kroese 2016] inference. Additionally (Section 5.3), we apply the Zar compiler backend to generate discrete uniform samplers that are proved free of modulo bias. Zar is implemented and fully verified in the Coq proof assistant.

### 1.1 Challenges

To understand the challenges, consider the 'primes' cpGCL program in Figure 1a, which computes a geometric posterior over the prime numbers as shown in Figure 1b. This program combines three fundamental features complicating inference: 1) nonuniform (biased) probabilistic choice, 2) unbounded loop-carried dataflow (a "non-i.i.d." loop [Kaminski 2019]), and 3) conditioning. The variable $b$ is drawn from a Bernoulli distribution with probability $p$ of "heads" (lines 2, 5). The variable $h$ (with initial value 0 and updated on line 4) counts the number of heads encountered before flipping tails. Finally, the terminal program state is conditioned on $h$ being prime (line 7).

*Eliminating Bias.* The probabilistic choices in the program of Figure 1a are specialized in Figure 1b to the nonuniform bias $p = \frac{2}{3}$. To obtain a sampler in the random bit model, the program must be transformed into a semantically equivalent one in which all choices have bias $\frac{1}{2}$. Moreover, probability expressions in cpGCL can be functions of the program state, so reduction of biased choices is not always possible via direct source-to-source translation (e.g., the probability expression on line 5 could depend on variable $h$). To address this state dependence and the use of nonuniform biases, we develop a new intermediate representation called choice-fix trees (Section 3.3). We compile cpGCL programs to the choice-fix representation and apply a debiasing transformation (Section 3.4) on choice-fix trees to generate samplers in the random bit model.

*Unbounded and Non-i.i.d. Loops.* The loop in Figure 1a is unbounded; it is not guaranteed to terminate within any fixed number of iterations, and can diverge (though with probability 0) when only heads are flipped. The tasks of sampling and inference are greatly complicated by the infinitary nature of unbounded loops, and thus much previous work on discrete PPs is limited to bounded loops [Chavira and Darwiche 2008; Holtzen et al. 2020, 2019; Huang et al. 2017]. Formal reasoning about infinitary computations requires substantial use of *coinduction* [Kozen and Silva 2017], which is notoriously difficult to use in proof assistants like Coq [Hur et al. 2013].

Moreover, the loop in Figure 1a is "non-i.i.d."; the update of counter variable $h$ on line 4 induces nontrivial data dependence between iterations of the loop, and consequently every value of $h \geq 0$ occurs with nonzero probability (the posterior has *infinite support*). Many interesting probabilistic programs such as the discrete Gaussian (see Section 5.4) exhibit such "loop-carried dependence" [Allen and Kennedy 1987]. Prior work on automated inference of unbounded loops and conditioning on observations in probabilistic programs has been restricted to the subclass of i.i.d. loops, i.e., those without loop-carried dependence [Bagnall et al. 2020].

Zar compiles the 'primes' program to an executable *interaction tree* (ITree) [Xia et al. 2020] formally guaranteed to produce samples from the geometric posterior shown in Fig 1b when provided uniform random bits from its environment (see Section 5.2 for empirical evaluation). The coinductive type of ITrees, while suitable for encoding potentially unbounded processes, is deceptively difficult to reason about formally. Coq's built-in mechanism for coinduction is often not sufficient [Chlipala 2013; Hur et al. 2013]. To facilitate reasoning on coinductive representations of samplers, we employ concepts from domain theory such as Scott-continuity [Abramsky and Jung 1994] and algebraic CPOs [Gunter 1993] (see related discussion in Section 3.5).

*Correctness of Samplers.* Verified compilers of conventional programming languages like C have somewhat well understood correctness guarantees (though see [Patterson and Ahmed 2019]). CompCert [Leroy 2009], for example, uses a simulation argument to prove a form of behavioral equivalence of source and target programs. Writing the specification of a compiler for a PPL is less straightforward. What does "behavioral equivalence" even mean when the result of the compilation pipeline is a probabilistic sampler that depends on a source of randomness?

A key idea of this paper is that the proof of a PPL compiler is essentially a reduction: as input, it takes a source of randomness (in our case, uniformly distributed random bits) and as output it produces a sampler on the posterior distribution generated by the conditional weakest pre-expectation semantics (cwp) of the program being compiled. We thus reduce the problem of sampling a program's posterior distribution to the comparatively simpler problem of sampling uniformly random bits. Making this reduction work formally means precisely characterizing the input source of randomness and the distributional correctness of the output sampler. We specify the input randomness in Section 4.1, drawing on the classic theory of uniform distribution modulo

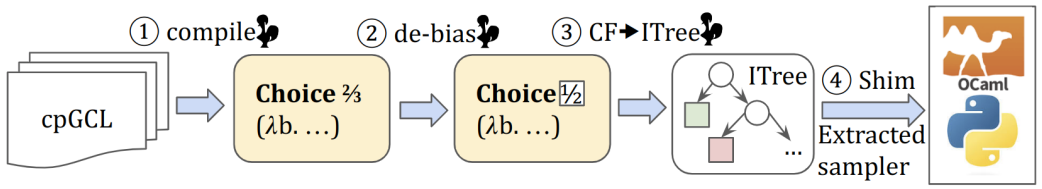



Fig. 2. Zar pipeline diagram showing (1) the compiler from cpGCL to CF trees (Section 3.1), (2) debiasing of probabilistic choices (Section 3.3), (3) generation of interaction trees from CF trees (Section 3.5), and (4) extraction for efficient execution in OCaml and Python (Section 5).

1 [Weyl 1916]. We characterize distributional correctness by proving that our samplers satisfy an *equidistribution theorem* (Section 4) wrt. the cwp semantics of source programs.

### 1.2 Contributions

**Concept.** We implement Zar: a formally verified compilation pipeline from discrete probabilistic programs with conditioning to proved-correct executable samplers in the random bit model, exploiting the key idea that all discrete distributions can be reduced to unbiased coin-flipping schemes [Knuth and Yao 1976; Saad et al. 2020b], and culminating in the *sampling equidistribution* theorem (Theorem 4.2) establishing correctness of compiled samplers. The entire system is fully implemented and verified in Coq.

**Technical.** The Zar system includes:

- a formalization of cpGCL and its associated cwp semantics (Section 2),
- an intermediate representation for cpGCL programs called *choice-fix* trees (Section 3.1), enabling optimizations and essential program transformations (e.g., elimination of redundant choices and reduction to the random bit model),
- a compiler pipeline (Section 3.3) from cpGCL to ITree samplers (Section 3.5),
- statement and proof of a general result establishing correctness of compiled samplers wrt. the cwp semantics of source programs, based on the notion of equidistribution (Section 4), and
- a Python 3 package for high-assurance uniform sampling (Section 5.3) as a thin wrapper around proved-correct samplers extracted from Coq.

**Evaluation.** We perform empirical validation of illustrative examples (Section 5) including the discrete Gaussian distribution (Section 5.4) and posterior inference over a simulated race between a hare and tortoise (Section 5.5, inspired by [Szymczak and Katoen 2020, Section 1]).

**Source Code.** Embedded hyperlinks in the PDF point to the underlying Coq sources The Python 3 package for uniform sampling is available in the Zar repository.

**Axiomatic Base.** We extend the type theory of Coq with excluded middle, indefinite description, and functional extensionality [Charguéraud 2017]. We also use Coq's standard real number library and a custom extensionality axiom for coinductive trees.

### 1.3 Current Limitations

Zar supports only discrete probabilistic cpGCL programs (which are naturally suited for many applications [Holtzen et al. 2020]) that terminate either absolutely or almost surely (i.e., with probability 1). Probabilities appearing in cpGCL programs must be rational numbers. We provide no guarantees regarding time/space or entropy usage (number of random bits required to obtain a sample), although we observe near entropy-optimality in some cases (cf. Section 5.3). We verify only the *compiler pipeline*. Verification of cpGCL programs using a program logic that is sound wrt. the cwp semantics is beyond the scope of this paper. Proofs on cpGCL programs wrt. their cwp

semantics can, however, be composed with our compiler correctness proofs (Theorems 3.7, 4.2) to obtain end-to-end guarantees on generated samplers. In future work, we plan to address the current limitation to discrete programs by extending cpGCL with sampling from continuous random variates over the unit interval, and to experiment with alternative sampling backends.

## 2 SYNTAX AND SEMANTICS OF cpGCL

This section presents cpGCL together with its conditional weakest pre-expectation semantics, cwp [Olmedo et al. 2018]. We extend cpGCL and cwp as follows:

(1) We add to cpGCL an additional command for drawing samples uniformly at random from a range of natural numbers (Section 2.1).
(2) We let probability expressions in choice commands depend on the program state (Section 2.1).
(3) We extend the weakest (liberal) expectation transformers wp (wlp) to better support reasoning about the probability of observation failure (Section 2.2).

### 2.1 Syntax of cpGCL

*Definition 2.1 (cpGCL).* Type cpGCL is defined inductively as:

$$\textsc{cpGCL-skip}\ \frac{}{\textbf{skip} : \text{cpGCL}} \qquad \textsc{cpGCL-assign}\ \frac{x : \text{ident} \qquad e : \Sigma \to \text{val}}{x \leftarrow e : \text{cpGCL}} \qquad \textsc{cpGCL-seq}\ \frac{c_1 : \text{cpGCL} \qquad c_2 : \text{cpGCL}}{c_1;\ c_2 : \text{cpGCL}}$$

$$\textsc{cpGCL-observe}\ \frac{e : \Sigma \to \mathbb{B}}{\textbf{observe}\ e : \text{cpGCL}} \qquad \textsc{cpGCL-ite}\ \frac{e : \Sigma \to \mathbb{B} \qquad c_1 : \text{cpGCL} \qquad c_2 : \text{cpGCL}}{\textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 : \text{cpGCL}}$$

$$\textsc{cpGCL-choice}\ \frac{p : \Sigma \to \mathbb{Q} \qquad \forall \sigma : \Sigma,\ 0 \le p\ \sigma \le 1 \qquad c_1 : \text{cpGCL} \qquad c_2 : \text{cpGCL}}{\{\ c_1\ \}\ [p]\ \{\ c_2\ \} : \text{cpGCL}}$$

$$\textsc{cpGCL-uniform}\ \frac{e : \Sigma \to \mathbb{N} \qquad \forall \sigma : \Sigma,\ 0 < e\ \sigma \qquad k : \mathbb{N} \to \text{cpGCL}}{\textbf{uniform}\ e\ k : \text{cpGCL}} \qquad \textsc{cpGCL-while}\ \frac{e : \Sigma \to \mathbb{B} \qquad c : \text{cpGCL}}{\textbf{while}\ e\ \textbf{do}\ c\ \textbf{end} : \text{cpGCL}}$$

The cpGCL (Definition 2.1) extends the guarded command language [Dijkstra 1975] with:

(1) **Probabilistic choice:** Given expression $e : \Sigma \to \mathbb{Q}$ such that $e\ \sigma \in [0, 1]$ for all program states $\sigma : \Sigma$, the command $\{\ c_1\ \}\ [e]\ \{\ c_2\ \}$ executes command $c_1$ with probability $e\ \sigma$, or $c_2$ with probability $1 - e\ \sigma$.
(2) **Conditioning:** Given predicate $P : \Sigma \to \mathbb{B}$ on program states, the command **observe** $P$ conditions the posterior distribution of the program on $P$.

Additionally, the command **uniform** $e\ k$ uniformly samples a natural number $0 \le n < e\ \sigma$ (where $\sigma$ is the current program state) and continues execution with command $k\ n$.

### 2.2 Conditional Weakest Pre-Expectation Semantics

We follow [Olmedo et al. 2018] in interpreting cpGCL programs using conditional weakest pre-expectation (cwp) semantics, a quantitative generalization of weakest precondition semantics [Dijkstra 1975]. Samplers produced by Zar are proved correct wrt. the cwp semantics of source programs.

An *expectation* is a function $f : \Sigma \to \mathbb{R}^{\infty}_{\ge 0}$ mapping program states to the nonnegative reals extended with $+\infty$. The cwp semantics interprets programs as expectation transformers: Given a *post-expectation* $f : \Sigma \to \mathbb{R}^{\infty}_{\ge 0}$ and program $c$ : cpGCL, the conditional weakest pre-expectation

cwp $c\ f : \Sigma \to \mathbb{R}^{\infty}_{\geq 0}$ is a function mapping program states $\sigma : \Sigma$ to the expected value of $f$ over the terminal states of $c$ given initial state $\sigma$ that are consistent with all observations.

Given a predicate $Q : \Sigma \to \mathbb{B}$ and program $c$ : cpGCL, cwp $c\ [Q] : \Sigma \to \mathbb{R}^{\infty}_{\geq 0}$ maps states $\sigma : \Sigma$ to the probability that $c$, when executed from initial state $\sigma$, terminates in a final state satisfying $Q$. Thus, cwp can be used to answer probabilistic queries about the execution behavior of programs. For more background on weakest pre-expectation semantics, see [Kaminski 2019, Chapters 2-4].

The cwp semantics is defined in terms of the more primitive weakest pre-expectation (wp) and weakest liberal pre-expectation (wlp) expectation transformers. Definitions 2.2 and 2.3 give generalized variants of wp and wlp respectively, extended with an additional Boolean parameter controlling how observation failure is handled (where $\mathbf{0}$ and $\mathbf{1}$ denote the constant expectations $\lambda\_.\ 0$ and $\lambda\_.\ 1$, respectively, and $F^n$ denotes the $n$-fold composition of functional $F$).

*Definition 2.2 ($\mathsf{wp}_\mathsf{b}$).* For $\mathsf{b} : \mathbb{B}$, $c$ : cpGCL, $f : \Sigma \to \mathbb{R}^{\infty}_{\geq 0}$, and $\sigma : \Sigma$, define $\mathsf{wp}_\mathsf{b}\ c\ f\ \sigma : \mathbb{R}^{\infty}_{\geq 0}$ by induction on $c$:

| $\mathsf{wp}_\mathsf{b} : \mathsf{cpGCL} \to (\Sigma \to \mathbb{R}^{\infty}_{\geq 0}) \to \Sigma \to \mathbb{R}^{\infty}_{\geq 0}$ | | |
|---|---|---|
| **skip** | $f$ | $\triangleq\ f$ |
| $x \leftarrow e$ | $f$ | $\triangleq\ f[x/e]$ |
| **observe** $e$ | $f$ | $\triangleq\ [e] \cdot f + [\neg e \wedge \mathsf{b}]$ |
| $c_1;\ c_2$ | $f$ | $\triangleq\ \mathsf{wp}_\mathsf{b}\ c_1\ (\mathsf{wp}_\mathsf{b}\ c_2\ f)$ |
| **if** $e$ **then** $c_1$ **else** $c_2$ | $f$ | $\triangleq\ [e] \cdot \mathsf{wp}_\mathsf{b}\ c_1\ f + [\neg e] \cdot \mathsf{wp}_\mathsf{b}\ c_2\ f$ |
| $\{\ c_1\ \}\ [p]\ \{\ c_2\ \}$ | $f$ | $\triangleq\ p \cdot \mathsf{wp}_\mathsf{b}\ c_1\ f + (\mathbf{1} - p) \cdot \mathsf{wp}_\mathsf{b}\ c_2\ f$ |
| **uniform** $e\ k$ | $f$ | $\triangleq\ \lambda\sigma.\ \frac{1}{e\,\sigma} \sum_{i=0}^{e\,\sigma-1} \mathsf{wp}_\mathsf{b}\ (k\ i)\ f\ \sigma$ |
| **while** $e$ **do** $c$ **end** | $f$ | $\triangleq\ \sup\ (F^n\ \mathbf{0})$, where |
| | | $F\ g \triangleq [e] \cdot \mathsf{wp}_\mathsf{b}\ c\ g + [\neg e] \cdot f$ |

*Definition 2.3 ($\mathsf{wlp}_\mathsf{b}$).* For $b : \mathbb{B}$, $c$ : cpGCL, $f : \Sigma \to \mathbb{R}^{\leq 1}_{\geq 0}$ (a bounded expectation), and $\sigma : \Sigma$, define $\mathsf{wlp}_\mathsf{b}\ c\ f\ \sigma : \mathbb{R}^{\leq 1}_{\geq 0}$ by induction on $c$ (showing only the **while** case as the rest are like wp, *mutatis mutandis*):

| $\mathsf{wlp}_\mathsf{b} : \mathsf{cpGCL} \to (\Sigma \to \mathbb{R}^{\leq 1}_{\geq 0}) \to \Sigma \to \mathbb{R}^{\leq 1}_{\geq 0}$ | | |
|---|---|---|
| **while** $e$ **do** $c$ **end** | $f$ | $\triangleq\ \inf\ (F^n\ \mathbf{1})$, where |
| | | $F\ g \triangleq [e] \cdot \mathsf{wlp}_\mathsf{b}\ c\ g + [\neg e] \cdot f$ |

We often omit the subscript when $b = \mathbf{false}$ as $\mathsf{wp}_{\mathbf{false}}$ ($\mathsf{wlp}_{\mathbf{false}}$) coincides with the classic definition of wp (wlp). The sup (inf) operation is defined wrt. the pointwise lifting to expectations of the standard order on $\mathbb{R}^{\infty}_{\geq 0}$ (i.e., $f \sqsubseteq g \iff \forall \sigma, f\ \sigma \leq g\ \sigma$ for expectations $f$ and $g$). The parameter b controls whether or not to include the probability mass of observation failure, so that we have $\mathsf{wp}_\mathsf{b}\ c\ f + \mathsf{wlp}_{\neg\mathsf{b}}\ c\ (\mathbf{1} - f) = \mathbf{1}$, where $f \sqsubseteq \mathbf{1}$ (the *invariant sum* property). The fundamental difference between wp and wlp can be characterized as follows:

• wp encodes *total* program correctness. When posing a query over predicate $Q$ using wp, we are asking "what is the probability that the program terminates *and* does so in a state satisfying $Q$?". Divergent execution paths (those which never terminate) contribute nothing to the pre-expectation.

• wlp encodes *partial* program correctness. When posing a query over predicate $Q$ using wlp, we are asking "what is the probability that the program either diverges *or* terminates in a state satisfying $Q$?". Divergent paths contribute their full probability mass to the weakest liberal pre-expectation.

Furthermore, wlp is defined only on *bounded expectations* $f : \Sigma \to \mathbb{R}^{\leq 1}_{\geq 0}$ as it is only meaningful for probabilities. It follows that wp and wlp coincide for bounded expectations on terminating programs (whether absolutely or almost surely).

The conditional weakest pre-expectation of expectation $f : \Sigma \to \mathbb{R}^{\infty}_{\geq 0}$ with respect to program $c$ : cpGCL is then defined following the approach of [Olmedo et al. 2018]:

*Definition 2.4 (cwp).* For $c$ : cpGCL and $f : \Sigma \to \mathbb{R}^{\infty}_{\geq 0}$,

$$\mathrm{cwp}\ c\ f : \Sigma \to \mathbb{R}^{\infty}_{\geq 0} \triangleq \frac{\mathrm{wp}_{\mathbf{false}}\ c\ f}{\mathrm{wlp}_{\mathbf{false}}\ c\ \mathbf{1}}.$$

## 3 COMPILING cpGCL

We compile cpGCL commands $c$ to samplers that we prove correct (cf. Section 4) wrt. the cwp semantics of $c$. First, we give an intermediate representation (Sections 3.1, 3.2)—*choice-fix* (CF) tree—to which a command $c$ is compiled (Section 3.3). Next, the generated CF tree is "unfolded" into a coinductive *interaction tree* (Section 3.5) implementing a sampler from the posterior distribution denoted by $c$. Finally, correctness of the compiler pipeline is established by the semantics preservation theorem (Theorem 3.7).

### 3.1 Choice-Fix Trees

Choice-fix (CF) trees are named for their two nonleaf constructors: **Choice** nodes for probabilistic choice and **Fix** nodes for encoding loops.

*Definition 3.1 (CF trees).* Define the type of CF trees $\mathcal{T}^{cf}$ inductively as:

$$\textsc{cf-leaf}\ \frac{\sigma : \Sigma}{\mathbf{Leaf}\ \sigma : \mathcal{T}^{cf}} \qquad \textsc{cf-fail}\ \frac{}{\mathbf{Fail} : \mathcal{T}^{cf}} \qquad \textsc{cf-choice}\ \frac{p : \mathbb{Q} \qquad 0 \leq p \leq 1 \qquad k : \mathbb{B} \to \mathcal{T}^{cf}}{\mathbf{Choice}\ p\ k : \mathcal{T}^{cf}}$$

$$\textsc{cf-fix}\ \frac{\sigma : \Sigma \qquad e : \Sigma \to \mathbb{B} \qquad g : \Sigma \to \mathcal{T}^{cf} \qquad k : \Sigma \to \mathcal{T}^{cf}}{\mathbf{Fix}\ \sigma\ e\ g\ k : \mathcal{T}^{cf}}$$

**Leaf** $\sigma$ denotes the end of a program execution with terminal state $\sigma$. **Fail** denotes a program execution in which an observed predicate (via the **observe** command) is violated. **Choice** $p\,k$ represents a probabilistic binary choice between two subtrees where rational bias $p \in [0, 1]$ denotes the probability of "heads" or "going left" (and $1 - p$ the probability of "tails" or "going right").

Lastly, **Fix** $\sigma\ e\ g\ k$ encodes a loop with initial state $\sigma$, guard condition $e$, body generator $g$, and continuation $k$, and should be understood operationally as follows: Starting with initial CF tree **Leaf** $\sigma$, repeatedly extend the leaves of the tree constructed thus far via either the generating function $g$ (when $e\ \sigma$ = **true**) or continuation $k$ (when $e\ \sigma$ = **false**). That is, $\sigma$ is the initial state of the loop, $e$ is the guard condition of the loop, $g$ is the generating function of the body of the loop, and $k$ is the continuation of the program after exiting the loop. Figure 3 shows the CF tree representation of the 'primes' program from Figure 1a.

```
Choice p (λb₀.
  if b₀ then
    Fix {h ↦ 0, b ↦ true} (λσ. σ[b])
      (λσ. Choice p (λb′. if b′
        then Leaf (σ[h ↦ h + 1, b ↦ true])
        else Leaf (σ[h ↦ h + 1, b ↦ false])))
      (λσ. if σ h is prime then Leaf σ else Fail)
  else Fail)
```

Fig. 3. CF tree term representation of Prog. 1a.

### 3.2 CF Tree Semantics

The *inference* (or *twp*) semantics of CF trees is defined analogously to the cwp semantics of cpGCL. The expression $\mathrm{twp}_{\mathbf{false}}\ t\ f$ denotes the expected value of expectation $f$ over CF tree $t$. When $\mathrm{b} = \mathbf{true}$, twp includes the probability mass of observation failure (the **Fail** case of Definition 3.2).

*Definition 3.2 ($\mathrm{twp}_\mathrm{b}$).* For $t : \mathcal{T}^{cf}$ a CF tree and $f : \Sigma \to \mathbb{R}^\infty_{\geq 0}$ an expectation, define $\mathrm{twp}_\mathrm{b}\ t\ f : \mathbb{R}^\infty_{\geq 0}$ by induction on $t$:

| $\mathrm{twp}_\mathrm{b} : \mathcal{T}^{cf} \to (\Sigma \to \mathbb{R}^\infty_{\geq 0}) \to \mathbb{R}^\infty_{\geq 0}$ | | | |
|---|---|---|---|
| **Leaf** $\sigma$ | $f$ | $\triangleq$ | $f\ \sigma$ |
| **Fail** | _ | $\triangleq$ | [b] |
| **Choice** $p\ k$ | $f$ | $\triangleq$ | $p \cdot \mathrm{twp}_\mathrm{b}\ (k\ \mathbf{true})\ f + (1-p) \cdot \mathrm{twp}_\mathrm{b}\ (k\ \mathbf{false})\ f$ |
| **Fix** $\sigma_0\ e\ g\ k$ | $f$ | $\triangleq$ | $\sup\ (F^n\ \mathbf{0})\ \sigma_0$, where |
| | | | $F\ h\ \sigma \triangleq$ if $e\ \sigma$ then $\mathrm{twp}_\mathrm{b}\ (g\ \sigma)\ h$ else $\mathrm{twp}_\mathrm{b}\ (k\ \sigma)\ f$ |

The "liberal" variant $\mathrm{twlp}_\mathrm{b}\ t\ f$ of inference semantics denotes the expected value of expectation $f$ over CF tree $t$ plus the probability mass of divergence (and plus the mass of observation failure when $b = \mathbf{true}$). Only the **Fix** case is shown here.

*Definition 3.3 ($\mathrm{twlp}_\mathrm{b}$).* For $t : \mathcal{T}^{cf}$ a CF tree and $f : \Sigma \to \mathbb{R}^{\leq 1}_{\geq 0}$ a bounded expectation, define $\mathrm{twlp}_\mathrm{b}\ t\ f : \mathbb{R}^{\leq 1}_{\geq 0}$ by induction on $t$:

| $\mathrm{twlp}_\mathrm{b} : \mathcal{T}^{cf} \to (\Sigma \to \mathbb{R}^{\leq 1}_{\geq 0}) \to \mathbb{R}^{\leq 1}_{\geq 0}$ | | | |
|---|---|---|---|
| **Fix** $\sigma_0\ e\ g\ k$ | $f$ | $\triangleq$ | $\inf\ (F^n\ \mathbf{1})\ \sigma_0$, where |
| | | | $F\ h\ \sigma \triangleq$ if $e\ \sigma$ then $\mathrm{twlp}_\mathrm{b}\ (g\ \sigma)\ h$ else $\mathrm{twlp}_\mathrm{b}\ (k\ \sigma)\ f$ |

The conditional (tcwp) semantics for CF trees then matches cwp (Def. 2.4):

*Definition 3.4 (tcwp).* For $t : \mathcal{T}^{cf}$ and $f : \Sigma \to \mathbb{R}^\infty_{\geq 0}$,

$$\mathrm{tcwp}\ t\ f : \mathbb{R}^\infty_{\geq 0} \triangleq \frac{\mathrm{twp}_{\mathbf{false}}\ t\ f}{\mathrm{twlp}_{\mathbf{false}}\ t\ \mathbf{1}}.$$

Our intent is that the tcwp semantics of the CF tree representation of a cpGCL program $c$ should coincide exactly with the cwp semantics of $c$. The following section presents a semantics-preserving compiler from cpGCL to CF trees.

### 3.3 Compiling to CF Trees

A command $c : \mathrm{cpGCL}$ is compiled to a function $[\![c]\!] : \Sigma \to \mathcal{T}^{cf}$ mapping initial state $\sigma : \Sigma$ to the CF tree encoding the sampling semantics of $c$ starting from $\sigma$. The operator '$\gg\!=$' used for compiling sequenced commands denotes bind in the CF tree monad.

*Definition 3.5 ($[\![\cdot]\!]$).* For $c : \mathrm{cpGCL}$ a command and $\sigma : \Sigma$ a program state, define $[\![c]\!]\ \sigma : \mathcal{T}^{cf}$ by induction $c$:

| $[\![\cdot]\!] : \mathrm{cpGCL} \to \Sigma \to \mathcal{T}^{cf}$ | | | |
|---|---|---|---|
| **skip** | $\sigma$ | $\triangleq$ | **Leaf** $\sigma$ |
| $x \leftarrow e$ | $\sigma$ | $\triangleq$ | **Leaf** $\sigma[x \mapsto e\ \sigma]$ |
| **observe** $e$ | $\sigma$ | $\triangleq$ | if $e\ \sigma$ then **Leaf** $\sigma$ else **Fail** |
| $c_1;\ c_2$ | $\sigma$ | $\triangleq$ | $[\![c_1]\!]\ \sigma \gg\!= [\![c_2]\!]$ |
| **if** $e$ **then** $c_1$ **else** $c_2$ | $\sigma$ | $\triangleq$ | if $e\ \sigma$ then $[\![c_1]\!]\ \sigma$ else $[\![c_2]\!]\ \sigma$ |
| $\{\ c_1\ \}\ [e]\ \{\ c_2\ \}$ | $\sigma$ | $\triangleq$ | **Choice** $(e\ \sigma)\ (\lambda b.$ if $b$ then $[\![c1]\!]\ \sigma$ else $[\![c2]\!]\ \sigma)$ |
| **uniform** $e\ k$ | $\sigma$ | $\triangleq$ | $\mathrm{uniform\_tree}\ (e\ \sigma) \gg\!= \lambda n.\ [\![k\ n]\!]\ \sigma$ |
| **while** $e$ **do** $c$ **end** | $\sigma$ | $\triangleq$ | **Fix** $\sigma\ e\ [\![c]\!]$ **Leaf** |

The symbol uniform_tree denotes a generic construction for uniform distributions over a fixed range of natural numbers, the specification of which is captured by the following lemma:

Lemma 3.6 (Uniform tree correctness). *Let* $0 < n : \mathbb{N}$ *and* $f : \mathbb{N} \to \mathbb{R}^{\infty}_{\geq 0}$ *an* $\mathbb{R}^{\infty}_{\geq 0}$*-valued function on* $\mathbb{N}$*. Then,* $\mathsf{twp}_{\mathbf{false}}\ (\mathsf{uniform\_tree}\ n)\ f = \frac{1}{n} \sum_{i=0}^{n-1} f\ i$. □

By setting $f = [\lambda m.\ m = k]$, we obtain as an immediate consequence of Lemma 3.6 that $\mathsf{twp}_{\mathbf{false}}\ (\mathsf{uniform\_tree}\ n)\ [\lambda m.\ m = k] = \frac{1}{n}$ for all $k < n$, or in other words, that uniform_tree is truly uniformly distributed (and thus free of modulo bias). The compiler phase is then proved correct by the following theorem establishing the correspondence of the cwp semantics of cpGCL programs with the tcwp semantics of the CF trees generated from them.

Theorem 3.7 (CF tree compiler correctness). *Let* $c : \mathsf{cpGCL}$, $f : \Sigma \to \mathbb{R}^{\infty}_{\geq 0}$, *and* $\sigma : \Sigma$. *Then,*

$$\mathsf{tcwp}\ (\llbracket c \rrbracket\ \sigma)\ f = \mathsf{cwp}\ c\ f\ \sigma. \quad \square$$

### 3.4 Debiasing CF Trees

CF trees generated by Def. 3.5 may have arbitrary $p \in [0, 1]$ bias values at choice nodes. To move toward the (uniform) random bit model, we apply a bias-elimination transformation $\mathsf{debias} : \mathcal{T}^{cf} \to \mathcal{T}^{cf}$ that uses the uniform_tree construction described above to replace probabilistic choices by semantically equivalent fair coin-flipping schemes. The resulting CF trees have $p = \frac{1}{2}$ at all choice nodes (we say that such trees are *unbiased*). Figure 4 shows how a choice node with bias $\frac{2}{3}$ is reduced to an equivalent unbiased CF tree.

The algorithm for translating a '**choice** $p\ k$' node with rational bias $p = \dfrac{n}{d}$ and subtrees $t_1 = k\ \mathbf{true}$ and $t_2 = k\ \mathbf{false}$ goes as follows:

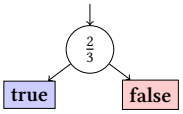
(1) Recursively translate $t_1$ and $t_2$, yielding $t_1'$ and $t_2'$ respectively,
(2) choose $m : \mathbb{N}$ such that $2^{m-1} < d \leq 2^m$,
(3) generate a perfect CF tree of depth $m$ with all terminal nodes marked by a special **loopback** value,
(4) replace the first $n$ terminals with copies of subtree $t_1'$, and the next $d$ terminals with copies of subtree $t_2'$, leaving $s^m - n - d$ **loopback** nodes remaining,
(5) coalesce duplicate leaf nodes to eliminate redundancy,
(6) wrap the tree in a **fix** constructor with guard condition that evaluates to true on the **loopback** value, and
(7) replace the original **choice** node with the newly generated tree.

In essence, the biased choice is replaced by a rejection sampler that simulates a biased coin by repeated flips of a fair one. An implementation of the choice translation algorithm is in the file 'uniform.v' under the name 'bernoulli_tree'. The overall debiasing transformation is then a straightforward recursive traversal of the input CF tree, using bernoulli_tree to replace biased **Choice** nodes with equivalent subtrees containing only unbiased choices.

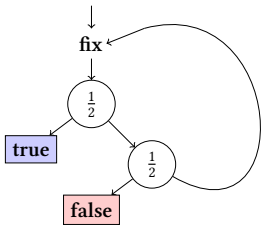
*Definition 3.8 (debias).* For $t : \mathcal{T}^{cf}$ a CF tree, define debias $t : \mathcal{T}^{cf}$ inductively on $t$:

| $\mathsf{debias} : \mathcal{T}^{cf} \to \mathcal{T}^{cf}$ | | |
|---|---|---|
| **Leaf** $\sigma$ | $\triangleq$ | **Leaf** $\sigma$ |
| **Fail** | $\triangleq$ | **Fail** |
| **Choice** $p\ k$ | $\triangleq$ | bernoulli_tree p $\gg\!=$ $\lambda$b. if b then debias (k **true**) else debias (k **false**) |
| **Fix** $\sigma\ e\ g\ k$ | $\triangleq$ | **Fix** $\sigma\ e$ (debias $\circ\ g$) (debias $\circ\ k$) |

The essential results for debias are: debias preserves tcwp semantics, and produces CF trees in which all choice nodes are unbiased.




(a) Biased choice with Pr(**true**) = $\frac{2}{3}$




(b) Debiased tree with Pr(**true**) = $\frac{2}{3}$

Fig. 4. **Choice** CF tree with $Pr(\textbf{true}) = \frac{2}{3}$ (left) and corresponding debiased CF tree (right).

Theorem 3.9 (debias is sound). *Let $t : \mathcal{T}^{cf}$ be a CF tree and $f : \Sigma \to \mathbb{R}^{\infty}_{\geq 0}$ an expectation. Then,*

$$\text{tcwp}\ (\text{debias}\ t)\ f = \text{tcwp}\ t\ f. \quad \square$$

Theorem 3.10 (debias produces unbiased trees). *Let $t : \mathcal{T}^{cf}$ be a CF tree. Then,*

$$p = \frac{1}{2} \text{ for every } \textbf{Choice}\ p\ k \text{ in debias } t. \quad \square$$

*The Need for Coinduction.* Debiased CF trees are close to being executable samplers in the random bit model. However, since they permit the existence of infinite execution paths (e.g., when $b = \textbf{true}$ ad infinitum in Program 1a), and hence denote sampling processes that can't be expected in general to terminate absolutely, we must first pass from the *inductive* CF tree encoding of samplers to an infinitary *coinductive* encoding.

### 3.5 Generating Interaction Trees

Interaction trees [Xia et al. 2020] (ITrees) are a general-purpose coinductive data structure for modeling effectful (co-)recursive programs that interact with their environments. The coq-itree library provides a suite of combinators for constructing ITrees along with a collection of principles for reasoning about their equivalence. An interaction tree computation performs an effect by raising an event (which may carry data) that is then handled by the environment, possibly providing data in return. This section shows how to generate executable ITree samplers from CF trees.

*ITree Syntax.* ITrees are parameterized by an event functor $E : \text{Type} \to \text{Type}$ that specifies the kinds of interactions the encoded process can have with its environment. In our case (Definition 3.11), there is only one kind of interaction: the sampler may query the environment for a single randomly generated bit. Thus the event functor boolE has a single constructor **GetBool** taking zero arguments, with type index $\mathbb{B}$ indicating that the environment's response should be a single bit.

*Definition 3.11 ($\mathcal{T}^{it}_A$).* Define $\mathcal{T}^{it}_A$ – the type of interaction trees with event functor boolE and element type $A$ – coinductively by:

$$\frac{}{\textbf{GetBool} : \text{boolE}\ \mathbb{B}}\ \text{BOOLE-GETBOOL} \qquad \frac{a : A}{\textbf{Ret}\ a : \mathcal{T}^{it}_A}\ \text{ITREE-RET} \qquad \frac{t : \mathcal{T}^{it}_A}{\textbf{Tau}\ t : \mathcal{T}^{it}_A}\ \text{ITREE-TAU} \qquad \frac{k : \mathbb{B} \to \mathcal{T}^{it}_A}{\textbf{Vis GetBool}\ k : \mathcal{T}^{it}_A}\ \text{ITREE-VIS}$$

Unfolding a CF tree to an interaction tree proceeds in two steps:
(1) Generating an ITree $t : \mathcal{T}^{it}_{(1+\Sigma)}$ by induction on the input CF tree, and then
(2) "tying the knot" on $t$ to produce the final ITree of type $\mathcal{T}^{it}_{\Sigma}$.

The LHS of the sum type $\mathbf{1} + \Sigma$ is used to encode observation failure. Since ITrees have just one kind of terminal constructor (**Ret**) to CF trees' two (**Leaf** and **Fail**), we translate **Fail** nodes to **Ret** (inl ()), and nodes of the form **Leaf** $x$ to **Ret** (inr $x$), where inl and inr are the left and right sum injections. **Fix** nodes are translated via application of the ITree.iter combinator (see [Xia et al. 2020, Section 4] on iteration with ITrees).

The first step is implemented by the function to_itree_open (see Figure 5a):

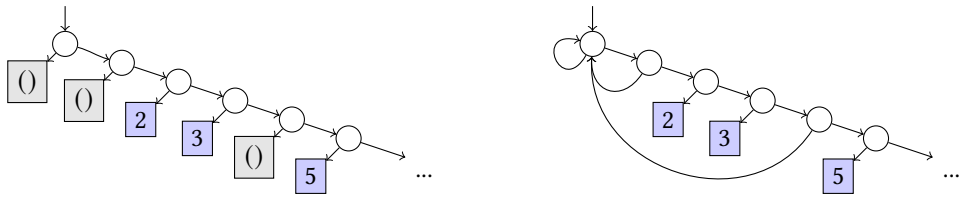
*Definition 3.12 (to_itree_open).* For unbiased CF tree $t : \mathcal{T}^{cf}$, define to_itree_open $t : \mathcal{T}^{it}_{1+\Sigma}$ by induction on $t$ as:

$$\begin{array}{lll}
\multicolumn{3}{l}{\text{to_itree_open} : \mathcal{T}^{cf} \to \mathcal{T}^{it}_{1+\Sigma}} \\
\textbf{Leaf}\ \sigma & \triangleq & \textbf{Ret}\ (\text{inr}\ \sigma) \\
\textbf{Fail} & \triangleq & \textbf{Ret}\ (\text{inl}\ ()) \\
\textbf{Choice}\ _\ k & \triangleq & \textbf{Vis}\ \textbf{GetBool}\ (\text{to_itree_open} \circ k) \\
\textbf{Fix}\ \sigma_0\ e\ g\ k & \triangleq & \text{ITree.iter}\ (\lambda\sigma.\ \text{if}\ e\ \sigma\ \text{then}\ y \leftarrow \text{to_itree_open}\ (g\ \sigma)\ ;; \\
 & & \qquad \text{match}\ y\ \text{with} \\
 & & \qquad \mid \text{inl}\ () \Rightarrow \textbf{Ret}\ (\text{inr}\ (\text{inl}\ ())) \\
 & & \qquad \mid \text{inr}\ \sigma' \Rightarrow \textbf{Ret}\ (\text{inl}\ \sigma') \\
 & & \qquad \text{end} \\
 & & \text{else ITree.map inr (to_itree_open}\ (k\ \sigma))\ \sigma_0
\end{array}$$

ITrees produced by to_itree_open treat observation failure as a terminal state with unit value () (Figure 5a). The following definition tie_itree corecursively "ties the knot" [Elkins 2021] through the left side of $\mathbf{1} + \Sigma$ via ITree.iter to produce an ITree rejection sampler that restarts from the beginning upon observation failure (Figure 5b).

*Definition 3.13 (tie_itree).* For $t : \mathcal{T}^{it}_{1+\Sigma}$, define tie_itree $t : \mathcal{T}^{it}_{\Sigma}$ as:

$$\text{tie\_itree}\ t \triangleq \text{ITree.iter}\ (\lambda\_.\ t)\ ()$$




(a) to_itree_open primes

(b) tie_itree (to_itree_open primes)

Fig. 5. Interaction trees generated by 'to_itree_open primes' (left) and then by "tying the knot" via 'tie_itree' (right), where 'primes' is the cpGCL program in Figure 1a.

*ITree semantics.* We wish to define an analogue itwp of the cwp semantics for ITree samplers and prove the correctness of ITree generation, i.e., that for all $t : \mathcal{T}^{cf}$ and $f : \Sigma \to \mathbb{R}^{\infty}_{\geq 0}$,

$$\text{itwp}\ f\ (\text{tie\_itree}\ (\text{to\_itree\_open}\ t)) = \text{tcwp}\ t\ f.$$

This turns out, however, to be difficult due to the lack of induction principle for ITrees. To overcome the problem, we note that ITree samplers form an *algebraic CPO* [Gunter 1993, Chapter 5], i.e., a domain in which all elements can be obtained as suprema of $\omega$-chains of finite approximations. Moreover, the types $\mathbb{R}^{\infty}_{\geq 0}$ of extended reals and $\mathbb{P}$ of propositions are CPOs. We exploit these

observations to provide a special kind of induction principle for coinductive trees (see [Gunter 1993, Lemma 5.24]). Such a principle enables the definition of Scott-continuous [Abramsky and Jung 1994] functions like itwp, and gives rise to a powerful suite of proof principles for reasoning about them via reduction to inductive proofs over inductive structures (for which Coq is much better suited than for coinduction). While details of this framework are outside the scope of this paper, they are implemented in the accompanying Coq sources.

*End-to-end compiler.* The compiler pipeline steps are composed via cpGCL_to_itree (where the function elim_choices reduces rationals and coalesces duplicate **Leaf** nodes) and proved correct by Theorem 3.15 (with positivity constraint on $\mathsf{wlp}_{\mathbf{false}}\ c\ \mathbf{1}\ \sigma$ assuring that the program doesn't condition on contradictory observations):

*Definition 3.14 (*cpGCL_to_itree*).* For $c$ : cpGCL and $\sigma : \Sigma$, define

$$\mathsf{cpGCL\_to\_itree}\ c\ \sigma \triangleq \mathsf{tie\_itree}\ (\mathsf{to\_itree\_open}\ (\mathsf{debias}\ (\mathsf{elim\_choices}\ (\mathsf{compile}\ c\ \sigma))))$$

Theorem 3.15 (Compiler Correctness). *Let* $c$ : cpGCL, $f : \Sigma \to \mathbb{R}^{\infty}_{\geq 0}$, *and* $\sigma : \Sigma$ *such that* $0 < \mathsf{wlp}_{\mathbf{false}}\ c\ \mathbf{1}\ \sigma$. *Then,*

$$\mathsf{cwp}\ c\ f\ \sigma = \mathsf{itwp}\ f\ (\mathsf{cpGCL\_to\_itree}\ c\ \sigma). \quad \square$$

Theorem 3.15 establishes semantics preservation of the compiler pipeline wrt. itwp, but doesn't directly guarantee properties of samples generated by the resulting ITrees. Drawing on basic measure theory [Halmos 2013] and the theory of uniform distribution modulo 1 [Kuipers and Niederreiter 2012; Weyl 1916], the next section extends the result of Theorem 3.15 to show that sequences of generated samples are equidistributed wrt. the cwp semantics of their source programs.

## 4 CORRECTNESS OF SAMPLING

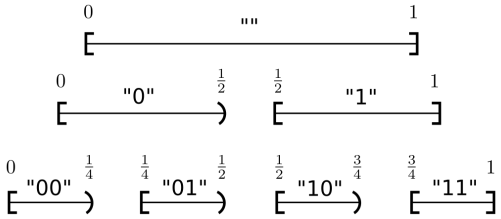
Given a suitable source of i.i.d. randomness (Section 4.1), a sampler for program $c$ : cpGCL is correct if it produces a sequence $x_n : \mathbb{N} \to \Sigma$ such that for any observation $Q : \Sigma \to \mathbb{P}$ over terminal program states, the *proportion of samples* falling within $Q$ asymptotically converges to the expected value of $[Q]$ (i.e., the probability of $Q$) according to $c$'s cwp semantics. In other words, a sampler is correct when the samples it produces are *equidistributed* [Becher and Grigorieff 2022] wrt. cwp.

This section formalizes the notion of equidistribution described above and proves the main sampling equidistribution theorem (Theorem 4.2, Section 4.3). We first clarify what is meant by "a suitable source of randomness" (Section 4.1) and then re-cast the problem of inference as that of computing a measure (Section 4.2).

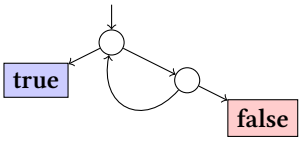
### 4.1 The Source of Randomness

We assume access to a stream of uniformly distributed bits. The *Cantor space* of countable sequences of bits (*bitstreams*), denoted $2^{\mathbb{N}}$, is modeled by the coinductive type Stream $\mathbb{B}$. We interpret samplers as measurable functions from $2^{\mathbb{N}}$ to the sample space $\Sigma$. To do so we first turn $2^{\mathbb{N}}$ into a measurable space by equipping it with a measure.

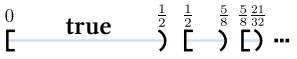
*Bisecting the Unit Interval.* To help visualize the measure on $2^{\mathbb{N}}$, consider (Figure 6a) the bisection scheme for identifying strings of bits (e.g., "0", "01", "011", etc.) with dyadic subintervals of the unit interval $[0, 1]$. Let $I(\omega)$ denote the interval corresponding to string $\omega$, and $B(\omega) \subseteq 2^{\mathbb{N}}$ the *basic set* of bitstreams with prefix $\omega$ (i.e., $B(\omega) = \{s \mid \omega \sqsubseteq s\}$ where '$\sqsubseteq$' is the prefix order). We arrange for the *measure* of $B(\omega)$ (denoted $\mu_\Omega(B(\omega))$) to be equal to the *length* of $I(\omega)$: exactly $2^{-n}$ where $n$ is the length of $\omega$. We then define the *source of randomness* to be the measure space $\Omega$ obtained by equipping $2^{\mathbb{N}}$ with measure $\mu_\Omega$ lifted to the Borel $\sigma$-algebra $\Sigma_\Omega$ of countable unions




(a) Bisection scheme identifying bitstrings with dyadic subintervals of $[0, 1]$




(b) ITree sampler $t_{\frac{2}{3}}$.




(c) Preimage intervals of event $\{\textbf{true}\}$ under $f_{t_{\frac{2}{3}}}$.

Fig. 6. Interval bisection scheme (left) and its application to ITree $t_{\frac{2}{3}}$ (right).

and complements of basic sets, coinciding with the standard Lebesgue measure $\lambda$ on the Borel $\sigma$-algebra generated by subintervals of $[0, 1]$.

### 4.2 Inference as Measure

We view ITree sampler $t : \mathcal{T}_\Sigma^{it}$ as a partial measurable function $f_t : \Omega \rightharpoonup \Sigma$ from $\Omega$ to the sample space $(\Sigma, \Sigma_\Sigma)$ where $\Sigma_\Sigma$ is the discrete (power set) $\sigma$-algebra on the space of program states $\Sigma$. Evaluation of $f_t$ on bitstream $s : 2^{\mathbb{N}}$ has two possible outcomes:

(1) The sampler *diverges*, consuming bits ad infinitum without ever producing a sample. In that case, we have $f_t(s) = \bot$, i.e., $f_t$ is *undefined* on $s$. We admit samplers for which such infinite executions are permitted but occur with probability 0 (i.e., the set $D \subseteq 2^{\mathbb{N}}$ of diverging inputs has measure 0). Or,
(2) a value $x$ is produced after consuming a finite prefix $\omega$ of $s$ corresponding to basic set $B(\omega)$. Thus, the function $f_t$ sends all bitstreams in $B(\omega)$ to output $x$.

For example, consider the ITree sampler $t_{\frac{2}{3}}$ in Figure 6b yielding **true** with probability $\frac{2}{3}$. The *preimage* set $f_{t_{\frac{2}{3}}}^{-1}(\{\textbf{true}\})$ of event $\{\textbf{true}\}$ under $f_{t_{\frac{2}{3}}}$ (i.e., the set of bitstreams sent by $f_{t_{\frac{2}{3}}}$ to **true**) has measure $\frac{2}{3}$. To see why, observe that $t_{\frac{2}{3}}$ contains infinitely many disjoint paths to **true** ("0", "100", "10100", etc.), with corresponding interval lengths $\frac{1}{2}$, $\frac{1}{8}$, $\frac{1}{32}$, etc., a geometric series with sum $\sum_k^\infty \frac{1}{2} \cdot \frac{1}{4}^k = \frac{\frac{1}{2}}{1-\frac{1}{4}} = \frac{2}{3}$.

We can exploit this observation to let the measure of any event $Q \subseteq \{\textbf{true}, \textbf{false}\}$ be equal to the measure of its preimage under $f_{t_{\frac{2}{3}}}$, thus inducing the following probability measure $\mu_{t_{\frac{2}{3}}} : \{\textbf{true}, \textbf{false}\} \to \mathbb{R}_{\geq 0}^{\leq 1}$ (where $\lambda(I)$ denotes the length of interval $I$):

$$
\begin{array}{llllll}
\mu_{t_{\frac{2}{3}}}(\emptyset) & = & \mu_\Omega(f_{t_{\frac{2}{3}}}^{-1}(\emptyset)) & = & \lambda(\emptyset) & = 0 \\
\mu_{t_{\frac{2}{3}}}(\{\textbf{true}\}) & = & \mu_\Omega(f_{t_{\frac{2}{3}}}^{-1}(\{\textbf{true}\})) & = & \lambda([0, \tfrac{2}{3})) & = \tfrac{2}{3} \\
\mu_{t_{\frac{2}{3}}}(\{\textbf{false}\}) & = & \mu_\Omega(f_{t_{\frac{2}{3}}}^{-1}(\{\textbf{false}\})) & = & \lambda([\tfrac{2}{3}, 1]) & = \tfrac{1}{3} \\
\mu_{t_{\frac{2}{3}}}(\{\textbf{true}, \textbf{false}\}) & = & \mu_\Omega(f_{t_{\frac{2}{3}}}^{-1}(\{\textbf{true}, \textbf{false}\})) & = & \lambda([0, 1]) & = 1
\end{array}
$$

*Computing Preimages.* We can generalize the above method to induce a probability measure $\mu_t : \Sigma \to \mathbb{R}^{\leq 1}_{\geq 0}$ from any $t : \mathcal{T}^{it}_{\Sigma}$ by letting $\mu_t(Q) = \mu_\Omega(f_t^{-1}(Q))$ (the *pushforward measure* of $Q$ under $f_t$), assigning to any event $Q : \Sigma \to \mathbb{P}$ a probability equal to the measure in $\Omega$ of its preimage under $f_t$. However, the preimage $f_t^{-1}(Q)$ is not easy to determine in general: it may be the union of infinitely many small intervals scattered throughout $[0, 1]$ in a complicated arrangement depending on the structure of $t$, where $t$ may be infinite and nonregular. We define $f_t^{-1}(Q)$ using the domain-theoretic machinery described in Section 3.5.

### 4.3 Equidistribution

To prove correctness of samplers generated by Zar, we show that any sequence of samples produced by them is *equidistributed* wrt. the cwp semantics of the programs they were compiled from. Our strategy is to assume uniform distribution of the source of randomness $\Omega$, and "push it forward" through the sampler to obtain the desired result. This section builds on the theory of uniform distribution modulo 1 (generalized to the class $\Sigma^0_1$ of countable unions of rational bounded intervals) adapted to collections of bitstreams.

*Uniform distribution of $\Omega$.* We assume access to a uniformly distributed sequence of bitstreams. But what does it mean for a sequence of bitstreams to be uniformly distributed? We cannot simply assert that any two bitstreams occur with equal probability because any particular bitstream occurs with probability zero and this may be true even for nonuniform distributions. Instead, we turn to a variation of the classic notion of "uniform distribution modulo 1", generalized to the class $\Sigma^0_1$ of subsets of $2^{\mathbb{N}}$ [Kuipers and Niederreiter 2012].

A subset $U \subseteq \mathcal{P}(2^{\mathbb{N}})$ is said to be $\Sigma^0_1$ when it is equal to $\bigcup_i^\infty B(\omega_i)$ for some countable collection $\{B(\omega_i)\}$ of basic sets. We remark that $f_t^{-1}(Q)$ is $\Sigma^0_1$ for all $Q : \Sigma \to \mathbb{P}$ and $t : \mathcal{T}^{it}_{\Sigma}$. The required notion of uniform distribution now follows:

*Definition 4.1 ($\Sigma^0_1$-u.d.).* A sequence $\{x_i\}$ of bitstreams is $\Sigma^0_1$-*uniformly distributed* ($\Sigma^0_1$-u.d.) when for every $U : \Sigma^0_1$, $\lim_{n\to\infty} \frac{1}{n} \sum_{i=0}^{n-1} [x_i \in U] = \mu_\Omega(U)$.

"Almost all" sequences of bitstreams are $\Sigma^0_1$-u.d. [Becher and Grigorieff 2022]. Moreover, $\Sigma^0_1$-u.d. has deep connections to Martin-Löf randomness [Martin-Löf 1966] and Schnorr randomness [Downey and Griffiths 2004] (cf. [Becher and Grigorieff 2022, Theorem 3]). We can now state the *equidistribution theorem*:

Theorem 4.2 (cpGCL equidistribution). *Let $c$ : cpGCL be a command, $\sigma : \Sigma$ a program state, $\{x_n\}$ a $\Sigma^0_1$-u.d. sequence of bitstreams, $Q : \Sigma \to \mathbb{P}$ a predicate over program states, and $t : \mathcal{T}^{it}_{\Sigma} =$* cpGCL_to_itree c *$\sigma$ the ITree sampler compiled from $c$. Then, the sequence $\{f_t(x_n)\}$ of samples is* cwp*-equidistributed wrt. $c$:*

$$\lim_{n\to\infty} \frac{1}{n} \sum_{i=0}^{n-1} [Q\ (f_t(x_i))] = \text{cwp}\ c\ [Q]\ \sigma. \quad \square$$

Theorem 4.2 is proved by reduction to an analogous theorem on ITrees. The basic idea is to show that cwp $C$ $[Q]$ $\sigma$ is equal to $\mu_\Omega(f_t^{-1}(Q))$ (i.e., that the probability of event Q according to cwp is equal to the measure of the preimage of $Q$ under $f_t$). The goal then follows from the assumption that $\{x_n\}$ is $\Sigma^0_1$-u.d. (by letting $U = f_t^{-1}(Q)$ in Definition 4.1).

## 5 EMPIRICAL VALIDATION

This section provides empirical validation of samplers compiled from cpGCL programs:

- *Correctness.* To validate Theorem 4.2, we compare the empirical distribution of generated samples with the expected true distribution wrt. total variation (TV) distance, Kullback-Leibler

```
let _ = Random.self_init ()  (* Seed PRNG. *)
let rec run t =              (* t : (boolE, 'a) itree *)
  match observe t with       (* Unfold itree. *)
  | RetF x -> x              (* Produce sample. *)
  | TauF t' -> run t'        (* Skip tau node. *)
  | VisF (_, k) ->           (* Consume random bit. *)
      run (k (Obj.magic (Random.bool ())))
```

Fig. 7. OCaml shim for execution of ITree samplers. The destructor 'observe' (not to be confused with the cpGCL command of the same name) unfolds the structure of the ITree $t$.

(KL) divergence [Kullback and Leibler 1951], and Symmetric Mean Absolute Percentage Error (SMAPE) [Armstrong 1985].

• *Performance.* Although generated samplers are not guaranteed to be entropy-optimal (in contrast to OPTAS [Saad et al. 2020b]), we measure statistics of the number of uniform random bits required to obtain a sample.

We do not verify the programs in this section wrt. their cwp semantics as we seek only to validate the correctness of the compilation pipeline.

*OCaml Shim.* All programs in this section are compiled to verified ITree samplers (as described in Section 3.5) and extracted to OCaml [Leroy et al. 2021; Letouzey 2008] for execution by the driver code in Figure 7. Thus, correctness of extracted samplers depends on the PRNG provided by the OCaml Random module being $\Sigma_1^0$-u.d. (Def. 4.1). Sample records are generated and written to disk for external analysis with handwritten Python code (see, e.g., /extract/die/analyze.py) and statistics routines provided by scipy.stats [SciPy 2022]).

*Trusted Computing Base.* Our TCB includes the Coq typechecker (and therefore the OCaml compiler and runtime), the specifications of cwp (Section 2.2) and equidistribution (Section 4.3), and the OCaml extraction mechanism and driver code in Figure 7.

*Empirical Evaluation.* The remainder of this section contains tables showing results of empirical evaluation of accuracy and entropy-performance of a collection of illustrative cpGCL programs (all wrt. a sample size of 100,000). In each table, the first column denotes the values taken by the parameter of the distribution (e.g., the bias parameter $p$ for Bernoulli, range $n$ for uniform, etc.). $\mu_x$ and $\sigma_x$ denote the mean and standard deviation of the posterior over variable $x$. TV, KL, and SMAPE denote the total variation distance, Kullback-Leibler (KL) divergence [Kullback and Leibler 1951], and symmetric mean absolute percentage error [Armstrong 1985], respectively, of the empirical distribution wrt. the true distribution. $\mu_{bit}$ and $\sigma_{bit}$ denote the mean and standard deviation of the number of uniform random bits required to obtain a sample.

*Entropy Usage.* The Shannon entropy [Shannon 1948] of a probability distribution provides a lower bound on the average number of uniformly random bits required to obtain a single i.i.d. sample. [Knuth and Yao 1976] show that an "entropy-optimal" sampler in the random bit model consumes no more than 2 bits on average than the entropy of the encoded distribution. Our samplers are not guaranteed to be entropy optimal (a direction for future work).

```
duel (p : ℚ) :=
  a ← false; b ← false;
  while a = b do flip a p; flip b p; end
```

(a) Dueling coins program with bias $p \in (0, 1)$.

```
die (n : ℕ) :=
  uniform n (λm. x ← m + 1)
```

(b) Rolling an $n$-sided die.

Fig. 8. Dueling coins (left) and n-sided die (right) cpGCL programs.

Table 1. Accuracy and entropy usage for Prog. 8a with $p = \frac{2}{3}$, $\frac{4}{5}$, and $\frac{1}{20}$. $mu_{bit}$ and $\sigma_{bit}$ increase as $p$ is goes further from $\frac{1}{2}$ due to increasing Shannon entropy of Bernoulli($p$).

| $p$ | $\mu_a$ | $\sigma_a$ | TV | KL | SMAPE | $\mu_{bit}$ | $\sigma_{bit}$ |
|---|---|---|---|---|---|---|---|
| 2/3 | 0.50 | 0.50 | $2.02\times10^{-3}$ | $1.20\times10^{-5}$ | $2.02\times10^{-3}$ | 12.0 | 9.39 |
| 4/5 | 0.50 | 0.50 | $2.16\times10^{-3}$ | $1.30\times10^{-5}$ | $2.16\times10^{-3}$ | 27.59 | 23.49 |
| 1/20 | 0.50 | 0.50 | $2.83\times10^{-3}$ | $2.30\times10^{-5}$ | $2.83\times10^{-3}$ | 134.97 | 129.07 |

Table 2. Accuracy and entropy usage for Prog. 1a with $p = \frac{1}{2}$, $\frac{2}{3}$, and $\frac{1}{5}$. $\mu_{bit}$ and $\sigma_{bit}$ are high when $p = \frac{1}{5}$ due to low probability of '$h$ is prime', illustrating a general weakness (entropy waste) of our rejection samplers when conditioning on low-probability events.

| $p$ | $\mu_h$ | $\sigma_h$ | TV | KL | SMAPE | $\mu_{bit}$ | $\sigma_{bit}$ |
|---|---|---|---|---|---|---|---|
| 1/2 | 2.64 | 1.10 | $2.33\times10^{-3}$ | $6.40\times10^{-5}$ | $7.63\times10^{-2}$ | 9.66 | 7.21 |
| 2/3 | 3.24 | 1.93 | $2.48\times10^{-3}$ | $1.10\times10^{-4}$ | $4.12\times10^{-2}$ | 25.31 | 20.59 |
| 1/5 | 2.19 | 0.44 | $7.44\times10^{-4}$ | $5.0\times10^{-6}$ | $5.19\times10^{-3}$ | 142.51 | 132.70 |

*Flip.* The command flip $x\ p$ : cpGCL performs a probabilistic choice (i.e., "flips a coin") with probability $p : \mathbb{Q}$ of **true** (or "heads") and assigns the result to variable $x$.

*Definition 5.1 (flip).* For $x$ : ident and $p \in [0, 1] \subseteq \mathbb{Q}$, define flip $x\ p$ as:

$$\text{flip } x\ p \triangleq \{\ x \leftarrow \textbf{true}\ \}\ [p]\ \{\ x \leftarrow \textbf{false}\ \}$$

### 5.1 Dueling Coins

Figure 8a illustrates an i.i.d. loop (unbounded but with no loop-carried dependence) simulating a fair coin using a biased one. The posterior distribution over $a$ is Bernoulli($\frac{1}{2}$) for any input bias $p \in (0, 1)$. The dueling coins illustrate a situation in which the average number of bits required to obtain a sample ($\mu_{bit} \sim 12$ when $p = \frac{2}{3}$ and $\mu_{bit} \sim 135$ when $p = \frac{1}{20}$, see Table 1) substantially exceeds the entropy (exactly 1) of the posterior.

### 5.2 Geometric Primes

Figure 1a illustrates the use of a *non-i.i.d.* loop and conditioning as follows: Repeatedly flip a coin with bias $p$, counting the number of heads until landing one tails. Finally, observe that the number of heads counted is a prime number. What, then, is the posterior over the number of heads $h$? The true posterior over prime $h$ is given by the probability mass function (pmf): $\Pr(X = h \mid h \text{ is prime}) = \frac{(1-p)^{h+1}}{\sum_{k\in\mathcal{P}} (1-p)^{k+1}}$, where $\mathcal{P}$ denotes the set of prime numbers. Table 2 shows accuracy and entropy statistics of the corresponding sampler compiled by Zar.

Table 3. Accuracy and entropy usage for Prog. 8b with $n = 6$, 200, and 10k (with Shannon entropies 2.59, 7.64, and 13.29, respectively). $\mu_{bit}$ and $\sigma_{bit}$ show good performance with near entropy-optimality.

| $n$ | $\mu_h$ | $\sigma_h$ | TV | KL | SMAPE | $\mu_{bit}$ | $\sigma_{bit}$ |
|---|---|---|---|---|---|---|---|
| 6 | 3.49 | 1.71 | $3.86\times10^{-3}$ | $5.80\times10^{-5}$ | $3.87\times10^{-3}$ | 3.66 | 1.33 |
| 200 | 100.42 | 57.65 | $1.77\times10^{-2}$ | $1.36\times10^{-3}$ | $1.77\times10^{-2}$ | 9.01 | 2.18 |
| 10k | 5011.87 | 2892.0 | $1.24\times10^{-1}$ | $7.33\times10^{-2}$ | $1.28\times10^{-1}$ | 15.62 | 2.74 |

Table 4. Comparison of 200-sided die samplers with output $x$. $T_{init}$ denotes time elapsed over construction and initializion of the sampler, and $T_s$ the total time to generate 100k samples.

| | $\mu_x$ | $\sigma_x$ | TV | KL | SMAPE | $\mu_{bit}$ | $\sigma_{bit}$ | $T_{init}$ | $T_s$ |
|---|---|---|---|---|---|---|---|---|---|
| Zar (OCaml) | 99.43 | 57.73 | $1.91\times10^{-2}$ | $1.75\times10^{-3}$ | $2.41\times10^{-2}$ | 9.0 | 2.16 | <1ms | 105ms |
| Zar (Py) | 99.87 | 57.63 | $1.95\times10^{-2}$ | $1.03\times10^{-3}$ | $2.20\times10^{-2}$ | 9.01 | 2.19 | <1ms | 292ms |
| FLDR (C) | 99.39 | 57.79 | $1.96\times10^{-2}$ | $1.18\times10^{-3}$ | $2.21\times10^{-2}$ | 9.01 | 2.18 | <1ms | 6ms |
| FLDR (Py) | 99.32 | 57.70 | $2.08\times10^{-2}$ | $1.36\times10^{-3}$ | $2.33\times10^{-2}$ | 9.0 | 2.16 | <1ms | 290ms |
| OPTAS (C) | 99.50 | 57.74 | $1.85\times10^{-2}$ | $1.20\times10^{-3}$ | $2.10\times10^{-2}$ | 8.55 | 1.27 | 3ms | 5ms |
| OPTAS (Py) | 99.58 | 57.69 | $2.12\times10^{-2}$ | $1.37\times10^{-3}$ | $2.37\times10^{-2}$ | 8.55 | 1.27 | 15ms | 330ms |

### 5.3 Uniform Sampling

Prog. 8b illustrates a program for rolling an n-sided die. Table 3 shows accuracy and entropy usage for $n = 6$, 200, and 10000.

*Comparison with FLDR and OPTAS.* The Fast Loaded Dice Roller (FLDR) [Saad et al. 2020a] is a time- and space-efficient algorithm for rolling an $n$-sided die, with implementations available in Python and C. Related to FLDR is OPTAS [Saad et al. 2020b], a system for optimal approximate sampling from discrete distributions wrt. a user-specified number of random bits, also with implementations in Python and C. Table 4 shows a comparison of a 200-sided die in FLDR and OPTAS (with 32-bit precision and the "hellinger" kernel) with OCaml and Python implementations of the 200-sided die based on a variant of uniform_tree from Section 3.3 that uses binary-encoded integers rather than unary natural numbers. Initialization time is negligible for both Zar and FLDR.

Zar *and TensorFlow 2.* We provide a Python 3 package (built from extracted samplers using pythonlib [pythonlib 2022]) exposing a simple interface for generating samples from verified uniform samplers. To demonstrate Zar's use as a high-assurance replacement for unverified samplers, we implement a TensorFlow 2 [Raschka and Mirjalili 2019] project (/python/tf/ in the source directory) for training an MNIST [LeCun et al. 1998] classifier via stochastic gradient descent. We observe a negligible effect on training performance and excellent accuracy on the test set.

### 5.4 Discrete Laplace and Gaussian

We define discrete variants of Laplace and Gaussian distributions (based on [Canonne et al. 2020]) as reusable subroutines for larger cpGCL programs (e.g., the hare and tortoise program in Section 5.5). These subroutines differ from flip in Def. 5.1 by making use of local variables. Although cpGCL lacks built-in support for procedure calls (which can be done in principle, as in [Olmedo et al. 2016]), they can be shallowly embedded if we take careful account of variables modified (i.e, "clobbered") within subroutines.

*5.4.1 Discrete Laplace.* A discrete analogue $\mathrm{Lap}_{\mathbb{Z}}(b)$ [Canonne et al. 2020] of the Laplace distribution (useful for, e.g., differential privacy [Ghosh et al. 2009], and as a subroutine for the discrete

```
laplace (out : ident) (s t : ℕ) :=
  lp ← true;
  while lp do
    uniform t (λu.
      bernoulli_exponential d (λs. u/t);
      if d then
        v ← 0; bernoulli_exponential il 1;
        while il do v ← v + 1; bernoulli_exponential il 1 end;
          x ← u + t · v; y ← x/s; flip c 1/2;
          if c ∧ y = 0 then skip
          else lp ← false; out ← (1 − 2 · [c]) · y
      else skip)
  end
```

Fig. 9. Sampling from $\mathsf{Lap}_{\mathbb{Z}}$. Modified variables: $k$, $a$, $i$, $b$, $lp$, $d$, $v$, $il$, $x$, $y$, and $c$. Variables $lp$ and $il$ ("loop" and "inner loop") are used for control flow. See [Canonne et al. 2020] for explanation and proof-of-correctness of the sampling algorithm.

| $s, t$ | $\mu_{out}$ | $\sigma_{out}$ | TV | KL | SMAPE | $\mu_{bit}$ | $\sigma_{bit}$ |
|---|---|---|---|---|---|---|---|
| 1, 2 | $1.79\times10^{-2}$ | 2.81 | $3.51\times10^{-3}$ | $4.20\times10^{-4}$ | $1.64\times10^{-1}$ | 10.47 | 7.04 |
| 2, 1 | $1.79\times10^{-3}$ | 0.60 | $1.47\times10^{-3}$ | $7.10\times10^{-5}$ | $5.30\times10^{-2}$ | 9.77 | 8.17 |
| 5, 2 | $-8.50\times10^{-4}$ | 0.44 | $1.24\times10^{-3}$ | $1.09\times10^{-4}$ | $1.37\times10^{-1}$ | 15.53 | 12.38 |

Fig. 10. Accuracy and entropy usage for Figure 9 with scale parameter $\frac{t}{s}$.

Gaussian in the following section) with scale parameter $b$ is defined by the probability mass function $\Pr_{\mathsf{Lap}_{\mathbb{Z}}(b)}(X = x) = \frac{e^{1/b}-1}{e^{1/b}+1} \cdot e^{-|x|/b}$. Figure 9 shows a cpGCL program for sampling from $\mathsf{Lap}_{\mathbb{Z}}(\frac{t}{s})$ for positive integers $s$ and $t$. The subroutine bernoulli_exponential, which takes parameter $\gamma$ and samples from a Bernoulli distribution with bias $e^{-\gamma}$, is defined in Appendix A.

*5.4.2 Discrete Gaussian.* A discrete analogue $\mathcal{N}_{\mathbb{Z}}(\mu, \sigma^2)$ [Canonne et al. 2020] of the Gaussian ("normal") distribution (useful for, e.g., lattice-based cryptography [Zhao et al. 2020], and as a subroutine for the hare-and-tortoise program in Section 5.5) with parameters $\mu$ and $\sigma$ is defined by the probability mass function:

$$\Pr_{\mathcal{N}_{\mathbb{Z}}(\mu,\sigma^2)}(X = x) = \frac{e^{-(x-\mu)^2/2\sigma^2}}{\sum_{y:\mathbb{Z}} e^{-(y-\mu)^2/2\sigma^2}}.$$

Figure 11 shows a cpGCL program for sampling from $\mathcal{N}_{\mathbb{Z}}(\mu, \sigma^2)$.

## 5.5 Hare and Tortoise

Our final example shown in Figure 13a illustrates the use of the discrete Gaussian subroutine along with a non-i.i.d. loop and conditioning to simulate a race between a hare and a tortoise along a one-dimensional line, and the use of Zar to perform Bayesian inference [Box and Tiao 2011]. The tortoise begins with uniformly-distributed head start $t_0$ and proceeds at a steady pace of 1 unit per time step. The hare begins at position 0 and occasionally (with probability $\frac{2}{5}$) leaps forward a Gaussian-distributed distance. The race ends when the hare reaches the tortoise, and then the terminal state is conditioned on predicate $P$. For example, by setting $P(time) = time \geq 10$ and

```
gaussian_0 (z : ident) (σ : ℚ) :=
  ol ← false;
  while ¬ol do laplace z 1 ⌊σ⌋ + 1; bernoulli_exponential ol (λs. ((|z| − σ²/t)²)/(2σ²)) end

gaussian (out : ident) (μ : Σ → ℤ) (σ : ℚ) :=
  gaussian out σ; out ← out + μ
```

Fig. 11. Sampling from $\mathcal{N}_{\mathbb{Z}}(\mu, \sigma^2)$. Note that the entropy usage depends only on $\sigma$ and not $\mu$. Modified variables: $k$, $a$, $i$, $b$, $lp$, $d$, $v$, $il$, $x$, $y$, $c$, $ol$, $z$. Variable $ol$ ("outer loop") is used for control flow. See [Canonne et al. 2020] for explanation and proof-of-correctness of the sampling algorithm.

| $\mu, \sigma$ | $\mu_z$ | $\sigma_z$ | TV | KL | SMAPE | $\mu_{bit}$ | $\sigma_{bit}$ |
|---|---|---|---|---|---|---|---|
| $0, 1$ | $-3.03\times10^{-3}$ | 1.0 | $2.71\times10^{-3}$ | $1.03\times10^{-4}$ | $4.49\times10^{-2}$ | 26.68 | 24.43 |
| $10, 2$ | 10.0 | 2.0 | $3.69\times10^{-3}$ | $1.16\times10^{-4}$ | $7.22\times10^{-2}$ | 37.61 | 29.10 |
| $-50, 5$ | $-50.01$ | 5.01 | $6.11\times10^{-3}$ | $4.46\times10^{-4}$ | $5.70\times10^{-2}$ | 43.66 | 31.20 |

Fig. 12. Accuracy and entropy usage for Figure 11 with mean $\mu$ and variance $\sigma^2$.

```
hare_tortoise (P : Σ → ℙ) :=
  uniform 10 (λn. t₀ ← n);
  tortoise ← t₀; hare ← 0; time ← 0;
  while hare < tortoise do
    time ← time + 1;
    tortoise → tortoise + 1;
    { gaussian jump 4 2;
      hare ← hare + jump } [2/5] { skip }
  end;
  observe P
```

(a) cpGCL program simulating a race between a hare and tortoise.

| $P$ | $\mu_{t_0}$ | $\sigma_{t_0}$ | $\mu_{bit}$ | $\sigma_{bit}$ |
|---|---|---|---|---|
| **true** | 4.49 | 2.87 | 193.88 | 220.06 |
| $time \leq 10$ | 3.80 | 2.79 | 273.87 | 378.82 |
| $time \geq 10$ | 6.18 | 2.31 | 596.68 | 359.85 |
| $time \geq 20$ | 6.40 | 2.25 | 1376.74 | 930.20 |

(b) Accuracy and entropy usage for Figure 13a. $\mu_{t_0}$ and $\sigma_{t_0}$ denote the mean and std deviation of the posterior over the tortoise's head start $t_0$, conditioned on $P$.

Fig. 13. Hare and tortoise cpGCL program (left) with accuracy and entropy statistics (right).

querying the posterior over $t_0$, we ask: "Given that it took at least 10 time steps for the hare to reach the tortoise, what are likely values for the tortoise's head start?" (see Figure 13b).

## 6 RELATED WORK

*Compilation.* [Holtzen et al. 2020, 2019] compile discrete probabilistic programs with bounded loops and conditioning to a symbolic representation based on binary decision diagrams (BDDs) [Akers 1978; Darwiche and Marquis 2002], exploiting independence of variables for efficient exact inference. Our CF trees are not as highly optimized as BDDs, and we currently do not support exact inference. However, while BDDs are suitable for representing *finite* Boolean functions, they are fundamentally insufficient for programs with unbounded loops for which no upper bound can be placed on the number of input bits per sample.

[Huang et al. 2017] compile PPs with continuous distributions (but not loops) to MCMC samplers for efficient approximate inference. MCMC algorithms generally provide better inference performance than Zar (which employs an "ordinary Monte Carlo" (OMC) strategy), but suffer from reliability issues (see Section 1). Zar is, to our knowledge, the only *formally verified* compiler for probabilistic programs with loops and conditioning.

*Verified Probabilistic Systems.* [Wang et al. 2021] implement a type system based on *guide types* to guarantee compatibility between model and guide functions in a PPL that compiles to Pyro [Bingham et al. 2019]. Pyro is more versatile than Zar, as it supports continuous distributions and programmable inference, but provides no formal guarantees on correctness of inference. [Selsam et al. 2018] implement Certigrad, a PP system for stochastic optimization with correctness guarantees in Lean [de Moura et al. 2015], but which does not support conditioning or inference.

*The Conditional Probabilistic Guarded Command Language.* The cpGCL and its corresponding cwp semantics were introduced by [Olmedo et al. 2018] and further developed by [Kaminski 2019] (including discussion of nondeterminism and expected running time) and [Szymczak and Katoen 2020] (adding support for continuous distributions). These works focus on using cwp as a program logic for verifying individual programs and metatheoretical properties of cpGCL, in contrast to Zar which focuses on verification of compilation to executable samplers.

*Interaction Trees.* Interaction trees have been used to verify compilation of an imperative programming language [Xia et al. 2020], networked servers [Koh et al. 2019; Letan and Régis-Gianas 2020], an HTTP key-value server [Zhang et al. 2021], and transactional objects [Lesani et al. 2022]. The Zar system presents a novel application of interaction trees to verified executable semantics of probabilistic programs, and employs a novel domain-theoretic framework for reasoning about them (see discussion in Section 3.5) based on the concept of algebraic CPO.

*Evaluation of PPLs.* [Dutta et al. 2018] implement a testing framework for PPLs called ProbFuzz that generates random test programs for various PPLs and attempts to detect irregularities in their inference results. We expect that Zar could be incorporated into ProbFuzz as a reference against which other discrete PPLs should be evaluated.

## 7 CONCLUSION

This paper presents the first formalization of cpGCL and its cwp semantics in a proof assistant, and implements Zar, the first formally verified compiler from a discrete PPL to proved-correct executable samplers. Zar uses a novel intermediate representation, CF trees, to optimize and debias probabilistic choices. CF trees are compiled to executable interaction trees encoding the sampling semantics of source programs in the random bit model. The full compilation pipeline is formally proved to satisfy an equidistribution theorem showing that the empirical distribution of generated samples converges to the true posterior distribution of the source cpGCL program. Zar's backend supports extraction to OCaml and has been used to generate samplers for a collection of probabilistic programs including the discrete Gaussian distribution.

**Acknowledgments.** We thank the anonymous reviewers and the paper's shepherd, Jan Hoffmann, for their comments. Banerjee's research was based on work supported by the National Science Foundation (NSF), while working at the Foundation. Any opinions, findings, and conclusions or recommendations expressed in this article are those of the authors and do not necessarily reflect the views of the NSF. Bagnall and Stewart were partially supported by NSF award #1657358.

**Data Availability Statement.** Sources for Zar and all examples in this paper are available on Zenodo with the identifier 10.5281/zenodo.7809333 [Bagnall et al. 2023].

## A SUBROUTINES FOR DISCRETE LAPLACE AND GAUSSIAN

This appendix contains cpGCL subroutines used in the discrete Laplace and Gaussian programs in Section 5.4.

*A.0.1 Inverse Exponential Bernoulli.* To sample from a discrete Laplace, we first require a subroutine for sampling from a Bernoulli distribution with inverse exponential bias. We begin with a routine (bernoulli_exponential_0_1) for the special case of $\mathbf{0} \le \gamma \le \mathbf{1}$, which modifies variables $k$ and $a$ (used as a counter and loop flag, respectively), followed by its generalization (bernoulli_exponential) to $\mathbf{0} \le \gamma$, additionally modifying variables $i$ and $b$ (also a counter and loop flag).

bernoulli_exponential_0_1 ($out$ : ident) ($\gamma : \Sigma \to \mathbb{Q}$) :=
  $k \leftarrow 0$; $a \leftarrow$ **true**;
  **while** $a$ **do** { $k \leftarrow k + 1$ } [$\frac{\gamma}{k+1}$] { $a \leftarrow$ **false**} **end**;
  **if** even $k$ **then** $out \leftarrow$ **true** **else** $out \leftarrow$ **false** **end**

Fig. 14. Sampling from Bernoulli(exp(−$\gamma$)), where $\mathbf{0} \le \gamma \le \mathbf{1}$

bernoulli_exponential ($out$ : ident) ($\gamma : \Sigma \to \mathbb{Q}$) :=
  **if** $\gamma \le 1$
  **then** bernoulli_exponential_0_1 $out$ $\gamma$
  **else** $i \leftarrow 1$; $b \leftarrow$ **true**;
    **while** $b \wedge i \le \gamma$ **do** bernoulli_exponential_0_1 $b$ **1**; $i \leftarrow i + 1$ **end**;
    **if** $b$ **then** bernoulli_exponential_0_1 $out$ ($\gamma - \lfloor\gamma\rfloor$) **else** $out \leftarrow 0$ **end**

Fig. 15. Sampling from Bernoulli(exp(−$\gamma$)), where $\mathbf{0} \le \gamma$

| $\gamma$ | $\mu_{out}$ | $\sigma_{out}$ | TV | KL | SMAPE | $\mu_{bit}$ | $\sigma_{bit}$ |
|---|---|---|---|---|---|---|---|
| 1/2 | 0.61 | 0.49 | $1.86\times10^{-3}$ | $1.0\times10^{-5}$ | $1.95\times10^{-3}$ | 2.54 | 2.16 |
| 3/2 | 0.23 | 0.42 | $1.36\times10^{-3}$ | $8.0\times10^{-6}$ | $1.96\times10^{-3}$ | 3.84 | 3.59 |
| 10 | $9.0\times10^{-5}$ | $9.49\times10^{-3}$ | $4.50\times10^{-5}$ | $2.50\times10^{-5}$ | $1.65\times10^{-1}$ | 4.56 | 5.11 |

Fig. 16. Accuracy and entropy usage for Figure 15.

## REFERENCES


Samson Abramsky and Achim Jung. 1994. Domain Theory. In *Handbook of Logic in Computer Science*, S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum (Eds.). Vol. 3. Clarendon Press, 1–168. https://global.oup.com/academic/product/handbook-of-logic-in-computer-science-9780198537625

Sheldon B. Akers. 1978. Binary Decision Diagrams. *IEEE Trans. Computers* 27, 6 (1978), 509–516. https://doi.org/10.1109/TC.1978.1675141

Randy Allen and Ken Kennedy. 1987. Automatic Translation of Fortran Programs to Vector Form. *ACM Trans. Program. Lang. Syst.* 9, 4 (1987), 491–542. https://doi.org/10.1145/29873.29875

Diego F. Aranha, Felipe Rodrigues Novaes, Akira Takahashi, Mehdi Tibouchi, and Yuval Yarom. 2020. LadderLeak: Breaking ECDSA with Less than One Bit of Nonce Leakage. In *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM, 225–242. https://doi.org/10.1145/3372297.3417268

J. Scott Armstrong. 1985. *Long-Range Forecasting: From Crystal Ball to Computer.* John Wiley & Sons, New York.

Alexander Bagnall, Gordon Stewart, and Anindya Banerjee. 2020. Coinductive Trees for Exact Inference of Probabilistic Programs. In *LAFI 2020: Languages for Inference*.

Alexander Bagnall, Gordon Stewart, and Anindya Banerjee. 2023. *Formally Verified Samplers From Probabilistic Programs With Loops and Conditioning*. https://doi.org/10.5281/zenodo.7809333

Verónica Becher and Serge Grigorieff. 2022. Randomness and uniform distribution modulo one. *Inf. Comput.* 285, Part (2022), 104857. https://doi.org/10.1016/j.ic.2021.104857

Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul A. Szerlip, Paul Horsfall, and Noah D. Goodman. 2019. Pyro: Deep Universal Probabilistic Programming. *J. Mach. Learn. Res.* 20 (2019), 28:1–28:6. http://jmlr.org/papers/v20/18-403.html

George E. .P. Box and George C. Tiao. 2011. *Bayesian Inference in Statistical Analysis*. John Wiley & Sons.

Clément L Canonne, Gautam Kamath, and Thomas Steinke. 2020. The discrete gaussian for differential privacy. *Advances in Neural Information Processing Systems* 33 (2020), 15676–15688.

Arthur Charguéraud. 2017. *CoqAndAxioms*. https://github.com/coq/coq/wiki/CoqAndAxioms

Mark Chavira and Adnan Darwiche. 2008. On probabilistic inference by weighted model counting. *Artif. Intell.* 172, 6-7 (2008), 772–799. https://doi.org/10.1016/j.artint.2007.11.002

Adam Chlipala. 2013. *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press. http://mitpress.mit.edu/books/certified-programming-dependent-types

Adnan Darwiche and Pierre Marquis. 2002. A Knowledge Compilation Map. *J. Artif. Intell. Res.* 17 (2002), 229–264. https://doi.org/10.1613/jair.989

Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description). In *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9195)*, Amy P. Felty and Aart Middeldorp (Eds.). Springer, 378–388. https://doi.org/10.1007/978-3-319-21401-6_26

Edsger W. Dijkstra. 1975. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM* 18, 8 (1975), 453–457. https://doi.org/10.1145/360933.360975

Rodney G. Downey and Evan J. Griffiths. 2004. Schnorr randomness. *J. Symb. Log.* 69, 2 (2004), 533–554. https://doi.org/10.2178/jsl/1082418542

Saikat Dutta, Owolabi Legunsen, Zixin Huang, and Sasa Misailovic. 2018. Testing probabilistic programming systems. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). ACM, 574–586. https://doi.org/10.1145/3236024.3236057

Saikat Dutta, Wenxian Zhang, Zixin Huang, and Sasa Misailovic. 2019. Storm: program reduction for testing and debugging probabilistic programming systems. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 729–739. https://doi.org/10.1145/3338906.3338972

Derek Elkins. 2021. *Tying the Knot*. https://wiki.haskell.org/Tying_the_Knot

Charles Geyer. 2011. Introduction to Markov Chain Monte Carlo. *Handbook of markov chain monte carlo* 20116022 (2011), 45.

Arpita Ghosh, Tim Roughgarden, and Mukund Sundararajan. 2009. Universally utility-maximizing privacy mechanisms. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009*, Michael Mitzenmacher (Ed.). ACM, 351–360. https://doi.org/10.1145/1536414.1536464

Noah D. Goodman, Vikash Mansinghka, Daniel M. Roy, Kallista A. Bonawitz, and Joshua B. Tenenbaum. 2012. Church: a language for generative models. *CoRR* abs/1206.3255 (2012). arXiv:1206.3255 http://arxiv.org/abs/1206.3255

Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. 2014. Probabilistic programming. In *Proceedings of the on Future of Software Engineering, FOSE 2014, Hyderabad, India, May 31 - June 7, 2014*, James D. Herbsleb and Matthew B. Dwyer (Eds.). ACM, 167–181. https://doi.org/10.1145/2593882.2593900

Carl A. Gunter. 1993. *Semantics of programming languages - structures and techniques*. MIT Press.

Paul R Halmos. 2013. *Measure theory*. Vol. 18. Springer.

Steven Holtzen, Guy Van den Broeck, and Todd D. Millstein. 2020. Scaling exact inference for discrete probabilistic programs. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 140:1–140:31. https://doi.org/10.1145/3428208

Steven Holtzen, Todd D. Millstein, and Guy Van den Broeck. 2019. Symbolic Exact Inference for Discrete Probabilistic Programs. *CoRR* abs/1904.02079 (2019). arXiv:1904.02079 http://arxiv.org/abs/1904.02079

Daniel Huang, Jean-Baptiste Tristan, and Greg Morrisett. 2017. Compiling Markov chain Monte Carlo algorithms for probabilistic modeling. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 111–125. https://doi.org/10.1145/3062341.3062375

Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. 2013. The power of parameterization in coinductive proof. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, 193–206. https://doi.org/10.1145/2429069.2429093

Benjamin Lucien Kaminski. 2019. *Advanced weakest precondition calculi for probabilistic programs.* Ph. D. Dissertation. RWTH Aachen University, Germany. http://publications.rwth-aachen.de/record/755408

Donald E. Knuth and Andrew C. Yao. 1976. The Complexity of Nonuniform Random Number Generation. In *Algorithms and Complexity: New Directions and Recent Results*, Joseph F. Traub (Ed.). Academic Press.

Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, Wolf Honoré, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. 2019. From C to interaction trees: specifying, verifying, and testing a networked server. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, Assia Mahboubi and Magnus O. Myreen (Eds.). ACM, 234–248. https://doi.org/10.1145/3293880.3294106

Dexter Kozen and Alexandra Silva. 2017. Practical coinduction. *Math. Struct. Comput. Sci.* 27, 7 (2017), 1132–1152. https://doi.org/10.1017/S0960129515000493

Lauwerens Kuipers and Harald Niederreiter. 2012. *Uniform distribution of sequences.* Courier Corporation.

Solomon Kullback and Richard A Leibler. 1951. On information and sufficiency. *The annals of mathematical statistics* 22, 1 (1951), 79–86.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324. https://doi.org/10.1109/5.726791

Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. https://doi.org/10.1145/1538788.1538814

Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2021. *The OCaml system release 4.13: Documentation and user's manual.* Intern report. Inria. 1–876 pages. https://hal.inria.fr/hal-00930213

Mohsen Lesani, Li-yao Xia, Anders Kaseorg, Christian J. Bell, Adam Chlipala, Benjamin C. Pierce, and Steve Zdancewic. 2022. C4: verified transactional objects. *Proc. ACM Program. Lang.* 6, OOPSLA (2022), 1–31. https://doi.org/10.1145/3527324

Thomas Letan and Yann Régis-Gianas. 2020. FreeSpec: specifying, verifying, and executing impure computations in Coq. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, Jasmin Blanchette and Catalin Hritcu (Eds.). ACM, 32–46. https://doi.org/10.1145/3372885.3373812

Pierre Letouzey. 2008. Extraction in Coq: An Overview. In *Logic and Theory of Algorithms, 4th Conference on Computability in Europe, CiE 2008, Athens, Greece, June 15-20, 2008, Proceedings (Lecture Notes in Computer Science, Vol. 5028)*, Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe (Eds.). Springer, 359–369. https://doi.org/10.1007/978-3-540-69407-6_39

Per Martin-Löf. 1966. The Definition of Random Sequences. *Inf. Control.* 9, 6 (1966), 602–619. https://doi.org/10.1016/S0019-9958(66)80018-9

Federico Olmedo, Friedrich Gretz, Nils Jansen, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Annabelle McIver. 2018. Conditioning in Probabilistic Programming. *ACM Trans. Program. Lang. Syst.* 40, 1 (2018), 4:1–4:50. https://doi.org/10.1145/3156018

Federico Olmedo, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. 2016. Reasoning about recursive probabilistic programs. In *2016 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. IEEE, 1–10.

Daniel Patterson and Amal Ahmed. 2019. The next 700 compiler correctness theorems (functional pearl). *Proc. ACM Program. Lang.* 3, ICFP (2019), 85:1–85:29. https://doi.org/10.1145/3341689

pythonlib. 2022. *pythonlib.* https://github.com/janestreet/pythonlib

Sebastian Raschka and Vahid Mirjalili. 2019. *Python machine learning: Machine learning and deep learning with Python, scikit-learn, and TensorFlow 2.* Packt Publishing Ltd.

Reuven Y Rubinstein and Dirk P Kroese. 2016. *Simulation and the Monte Carlo method.* John Wiley & Sons.

Feras Saad, Cameron E. Freer, Martin C. Rinard, and Vikash Mansinghka. 2020a. The Fast Loaded Dice Roller: A Near-Optimal Exact Sampler for Discrete Probability Distributions. In *The 23rd International Conference on Artificial Intelligence and Statistics, AISTATS 2020, 26-28 August 2020, Online [Palermo, Sicily, Italy] (Proceedings of Machine Learning Research, Vol. 108)*, Silvia Chiappa and Roberto Calandra (Eds.). PMLR, 1036–1046. http://proceedings.mlr.press/v108/saad20a.html

Feras A. Saad, Cameron E. Freer, Martin C. Rinard, and Vikash K. Mansinghka. 2020b. Optimal approximate sampling from discrete probability distributions. *Proc. ACM Program. Lang.* 4, POPL (2020), 36:1–36:31. https://doi.org/10.1145/3371104

SciPy. 2022. *scipy.stats.* https://docs.scipy.org/doc/scipy/reference/stats.html

Kudelski Security. 2020. *The definitive guide to "Modulo Bias and how to avoid it"!* https://research.kudelskisecurity.com/2020/07/28/the-definitive-guide-to-modulo-bias-and-how-to-avoid-it/

Daniel Selsam, Percy Liang, and David L Dill. 2018. Formal methods for probabilistic programming. In *Workshop on Probabilistic Programming Languages, Semantics, and Systems.*

Claude E. Shannon. 1948. A mathematical theory of communication. *Bell Syst. Tech. J.* 27, 3 (1948), 379–423. https://doi.org/10.1002/j.1538-7305.1948.tb01338.x

Marcin Szymczak and Joost-Pieter Katoen. 2020. Weakest Preexpectation Semantics for Bayesian Inference. *CoRR* abs/2005.09013 (2020). arXiv:2005.09013 https://arxiv.org/abs/2005.09013

Di Wang, Jan Hoffmann, and Thomas W. Reps. 2021. Sound probabilistic inference via guide types. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 788–803. https://doi.org/10.1145/3453483.3454077

Hermann Weyl. 1916. Über die gleichverteilung von zahlen mod. eins. *Math. Ann.* 77, 3 (1916), 313–352.

Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2020. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.* 4, POPL (2020), 51:1–51:32. https://doi.org/10.1145/3371119

Jonathan S Yedidia, William T Freeman, and Yair Weiss. 2003. Understanding belief propagation and its generalizations. *Exploring artificial intelligence in the new millennium* 8 (2003), 236–239.

Hengchu Zhang, Wolf Honoré, Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. 2021. Verifying an HTTP Key-Value Server with Interaction Trees and VST. In *12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference) (LIPIcs, Vol. 193)*, Liron Cohen and Cezary Kaliszyk (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 32:1–32:19. https://doi.org/10.4230/LIPIcs.ITP.2021.32

Raymond K Zhao, Ron Steinfeld, and Amin Sakzad. 2020. COSAC: Compact and scalable arbitrary-centered discrete Gaussian sampling over integers. In *International Conference on Post-Quantum Cryptography*. Springer, 284–303.

Received 2022-11-10; accepted 2023-03-31