Guaranteed Bounds for Posterior Inference in Universal Probabilistic Programming

Raven Beutner*
CISPA Helmholtz Center for Information Security
Germany

C.-H. Luke Ong
University of Oxford
United Kingdom

Fabian Zaiser
University of Oxford
United Kingdom

Abstract

We propose a new method to approximate the posterior distribution of probabilistic programs by means of computing *guaranteed* bounds. The starting point of our work is an interval-based trace semantics for a recursive, higher-order probabilistic programming language with continuous distributions. Taking the form of (super-/subadditive) measures, these lower/upper bounds are non-stochastic and provably correct: using the semantics, we prove that the actual posterior of a given program is sandwiched between the lower and upper bounds (soundness); moreover, the bounds converge to the posterior (completeness). As a practical and sound approximation, we introduce a weight-aware interval type system, which automatically infers interval bounds on not just the return value but also the weight of program executions, simultaneously. We have built a tool implementation, called GuBPI, which automatically computes these posterior lower/upper bounds. Our evaluation on examples from the literature shows that the bounds are useful, and can even be used to recognise wrong outputs from stochastic posterior inference procedures.

CCS Concepts: • **Mathematics of computing** → **Probabilistic inference problems**; • **Theory of computation** → *Program analysis*; • **Software and its engineering** → **Formal methods**.

Keywords: probabilistic programming, Bayesian inference, verification, abstract interpretation, operational semantics, interval arithmetic, type system, symbolic execution

ACM Reference Format:
Raven Beutner, C.-H. Luke Ong, and Fabian Zaiser. 2022. Guaranteed Bounds for Posterior Inference in Universal Probabilistic Programming. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22), June 13–17, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3519939.3523721

*Supported by the Saarbrücken Graduate School of Computer Science.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
PLDI '22, June 13–17, 2022, San Diego, CA, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9265-5/22/06.
https://doi.org/10.1145/3519939.3523721

1 Introduction

Probabilistic programming is a rapidly developing discipline at the interface of programming and Bayesian statistics [31, 32, 60]. The idea is to express probabilistic models (incorporating the prior distributions) and the observed data as programs, and to use a general-purpose Bayesian inference engine, which acts directly on these programs, to find the posterior distribution given the observations.

Some of the most influential probabilistic programming languages (PPLs) used in practice are *universal* (i.e. the underlying language is Turing-complete); e.g. Church [30], Anglican [59], Gen [18], Pyro [6], and Turing [24]. Using stochastic branching, recursion, and higher-order features, universal PPLs can express arbitrarily complex models. For instance, these language constructs can be used to incorporate probabilistic context free grammars [41], statistical phylogenetics [50], and even physics simulations [3] into probabilistic models. However, expressivity of the PPL comes at the cost of complicating the posterior inference. Consider, for example, the following problem from [39, 40].

Example 1.1 (Pedestrian). A pedestrian has gotten lost on a long road and only knows that they are a random distance between 0 and 3 km from their home. They repeatedly walk a uniform random distance of at most 1 km in either direction, until they find their home. When they arrive, a step counter tells them that they have traveled a distance of 1.1 km in total. Assuming that the measured distance is normally distributed around the true distance with standard deviation 0.1 km, what is the posterior distribution of the starting point? We can model this with a probabilistic program:

$$\begin{array}{l}\textbf{let } \mathit{start} = 3 \times \textbf{sample } \mathrm{uniform}(0,1) \textbf{ in}\\ \textbf{letrec } \mathit{walk}\ x = \textbf{if } x \le 0 \textbf{ then } 0 \textbf{ else}\\ \quad \textbf{let } \mathit{step} = \textbf{sample } \mathrm{uniform}(0,1) \textbf{ in}\\ \quad \mathit{step} + \mathit{walk}\big((x+\mathit{step}) \oplus_{0.5} (x - \mathit{step})\big)\\ \textbf{let } \mathit{distance} = \mathit{walk}\ \mathit{start} \textbf{ in}\\ \textbf{observe } \mathit{distance} \textbf{ from } \mathrm{Normal}(1.1, 0.1);\\ \mathit{start}\end{array}$$

Here **sample** uniform(a, b) samples a uniformly distributed value in $[a, b]$, $\oplus_{0.5}$ is probabilistic branching, and **observe** M **from** D observes the value of M from distribution D.

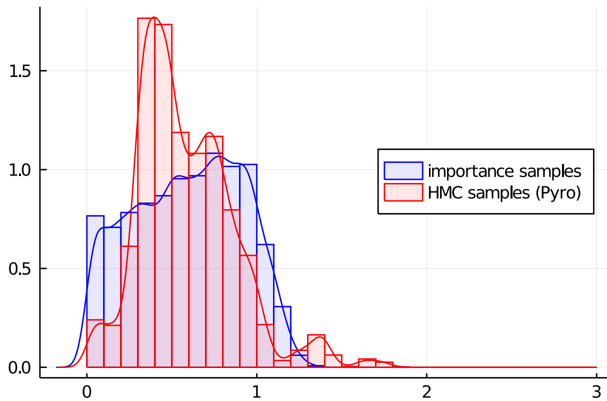


Figure 1. Histogram of samples from the posterior distribution of Example 1.1 and wrong samples produced by the probabilistic programming system Pyro.

Example 1.1 is a challenging model for inference algorithms in several regards: not only does the program use stochastic branching and recursion, but the number of random variables generated is unbounded – it's *nonparametric* [28, 36, 40]. To approximate the posterior distribution of the program, we apply two standard inference algorithms: likelihood-weighted importance sampling (IS), a simple algorithm that works well on low-dimensional models with few observations [47]; and Hamiltonian Monte Carlo (HMC) [21], a successful MCMC algorithm that uses gradient information to efficiently explore the parameter space of high-dimensional models. Figure 1 shows the results of the two inference methods as implemented in Anglican [59] (for IS) and Pyro [6] (for HMC): they clearly disagree! But how is the user supposed to know which (if any) of the two results is correct?

Note that *exact* inference methods (i.e. methods that try to compute a closed-form solution of the posterior inference problem using computer algebra and other forms of symbolic computation) such as PSI [25, 26], Hakaru [45], Dice [37], and SPPL [53] are only applicable to non-recursive models, and so they don't work for Example 1.1.

1.1 Guaranteed Bounds

The above example illustrates central problems with both approximate stochastic and exact inference methods. For approximate methods, there are no guarantees for the results they output after a finite amount of time, leading to unclear inference results (as seen in Fig. 1).[1] For exact methods, the symbolic engine may fail to find a closed-form description of the posterior distribution and, more importantly, they are only applicable to very restricted classes of programs (most notably, non-recursive models).

Instead of computing approximate or exact results, this work is concerned with computing *guaranteed* bounds on the posterior distribution of a probabilistic program. Concretely, given a probabilistic program P and a measurable set $U \subseteq \mathbb{R}$ (given as an interval), we infer upper and lower bounds on $[\![P]\!](U)$ (formally defined in Section 2), i.e. the posterior probability of P on U.[2] Such bounds provide a ground truth to compare approximate inference results with: if the approximate results violate the bounds, the inference algorithm has not converged yet or is even ill-suited to the program in question. Crucially, our method is applicable to arbitrary (and in particular recursive) programs of a universal PPL. For Example 1.1, the bounds computed by our method (which we give in Section 7) are tight enough to separate the IS and HMC output. In this case, our method infers that the results given by HMC are wrong (i.e. violate the guaranteed bounds) whereas the IS results are plausible (i.e. lie within the guaranteed bounds). To the best of our knowledge, no existing methods can provide such definite answers for programs of a universal PPL.

1.2 Contributions

The starting point of our work is an interval-based operational semantics [4]. In our semantics, we evaluate a program on *interval traces* (i.e. sequences of intervals of reals with endpoints between 0 and 1) to approximate the outcomes of sampling, and use interval arithmetic [19] to approximate numerical operations (Section 3). Our semantics is sound in the sense that any (compatible and exhaustive) set of interval traces yields lower and upper bounds on the posterior distribution of a program. These lower/upper bounds are themselves super-/subadditive measures. Moreover, under mild conditions (mostly restrictions on primitive operations), our semantics is also complete, i.e. for any $\epsilon > 0$ there exists a countable set of interval traces that provides ϵ-tight bounds on the posterior. Our proofs hinge on a combination of stochastic symbolic execution and the convergence of Riemann sums, providing a natural correspondence between our interval trace semantics and the theory of (Riemann) integration (Section 4).

Based on our interval trace semantics, we present a practical algorithm to automate the computation of guaranteed bounds. It employs an interval type system (together with constraint-based type inference) that bounds both the value of an expression in a refinement-type fashion *and* the score weight of any evaluation thereof. The (interval) bounds inferred by our type system fit naturally in the domain of our semantics. This enables a sound approximation of the

[1] Take MCMC sampling algorithms. Even though the Markov chain will eventually converge to the target distribution, we do not know how long to iterate the chain to ensure convergence [47, 51]. Likewise for variational inference [62]: given a variational family, there is no guarantee that a given value for the KL-divergence (from the approximating to the posterior distribution) is attainable by the minimising distribution.

[2] By repeated application of our method on a discretisation of the domain we can compute histogram-like bounds.

behaviour of a program with finitely many interval traces (Section 5).

We implemented our approach in a tool called GuBPI[3] (**Gu**aranteed **B**ounds for **P**osterior **I**nference), described in Section 6, and evaluate it on a suite of benchmark programs from the literature. We find that the bounds computed by GuBPI are competitive in many cases where the posterior could already be inferred exactly. Moreover, GuBPI's bounds are useful (in the sense that they are precise enough to rule out erroneous approximate results as in Fig. 1, for instance) for recursive models that could not be handled rigorously by any method before (Section 7).

1.3 Scope and Limitations

The contributions of this paper are of both theoretical and practical interest. On the theoretical side, our novel semantics underpins a sound and deterministic method to compute guaranteed bounds on program denotations. As shown by our completeness theorem, this analysis is applicable—in the sense that it computes arbitrarily tight bounds—to a very broad class of programs. On the practical side, our analyser GuBPI implements (an optimised version of) our semantics. As is usual for exact/guaranteed[4] methods, our semantics considers an exponential number of program paths, and partitions each sampled value into a finite number of interval approximations. Consequently, GuBPI generally struggles with high-dimensional models. We believe GuBPI to be most useful for unit-testing of implementations of Bayesian inference algorithms such as Example 1.1, or to compute results on (recursive) programs when non-stochastic, guaranteed bounds are needed.

2 Background

2.1 Basic Probability Theory and Notation

We assume familiarity with basic probability theory, and refer to [49] for details. Here we just fix the notation. A *measurable space* is a pair (Ω, Σ_Ω) where Ω is a set (of outcomes) and $\Sigma_\Omega \subseteq 2^\Omega$ is a σ-algebra defining the measurable subsets of Ω. A *measure* on (Ω, Σ_Ω) is a function $\mu : \Sigma_\Omega \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ that satisfies $\mu(\emptyset) = 0$ and is σ-additive. For $\mathbb{R}^n$, we write $\Sigma_{\mathbb{R}^n}$ for the Borel σ-algebra and λ_n for the Lebesgue measure on $(\mathbb{R}^n, \Sigma_{\mathbb{R}^n})$. The Lebesgue integral of a measurable function f with respect to a measure μ is written $\int f \,\mathrm{d}\mu$ or $\int f(x)\,\mu(\mathrm{d}x)$. Given a predicate ψ on Ω, we define the Iverson brackets $[\psi] : \Omega \to \mathbb{R}$ by mapping all elements that satisfy ψ to 1 and all others to 0. For $A \in \Sigma_\Omega$ we define the bounded integral $\int_A f \,\mathrm{d}\mu := \int f(x) \cdot [x \in A] \mu(\mathrm{d}x)$.

[3]GuBPI (pronounced "guppy") is available at gubpi-tool.github.io.

[4]By "exact/guaranteed methods", we mean inference algorithms that compute deterministic (non-stochastic) results about the mathematical denotation of a program. In particular, they are correct with probability 1, contrary to stochastic methods.

$$\frac{}{((\lambda x.M)V, s, w) \to (M[V/x], s, w)} \qquad \frac{}{(\mathsf{sample}, r\,s, w) \to (\underline{r}, s, w)}$$

$$\frac{}{((\mu_x^\varphi.M)V, s, w) \to (M[V/x, (\mu_x^\varphi.M)/\varphi], s, w)}$$

$$\frac{}{(\underline{f}(\underline{r_1}, \cdots, \underline{r_{|f|}}), s, w) \to (\underline{f(r_1, \cdots, r_{|f|})}, s, w)}$$

$$\frac{r \leq 0}{(\mathsf{if}(\underline{r}, N, P), s, w) \to (N, s, w)} \qquad \frac{r > 0}{(\mathsf{if}(\underline{r}, N, P), s, w) \to (P, s, w)}$$

$$\frac{r \geq 0}{(\mathsf{score}(\underline{r}), s, w) \to (\underline{r}, s, w \cdot r)} \qquad \frac{(R, s, w) \to (M, s', w')}{(E[R], s, w) \to (E[M], s', w')}$$

Figure 2. Standard (CbV) reduction rules for SPCF ($\to$).

2.2 Statistical PCF (SPCF)

As our probabilistic programming language of study, we use *statistical PCF* (SPCF) [39], a typed variant of [8]. SPCF includes primitive operations which are measurable functions $f : \mathbb{R}^{|f|} \to \mathbb{R}$, where $|f| \geq 0$ denotes the arity of the function. *Values* and *terms* of SPCF are defined as follows:

$$\begin{aligned} V &:= x \mid \underline{r} \mid \lambda x.M \mid \mu_x^\varphi.M \\ M, N, P &:= V \mid MN \mid \mathsf{if}(M, N, P) \mid \underline{f}(M_1, \cdots, M_{|f|}) \\ &\quad\mid \mathsf{sample} \mid \mathsf{score}(M) \end{aligned}$$

where x and φ are variables, f is a primitive operation, and $\underline{r}$ a constant with $r \in \mathbb{R}$. Note that we write $\mu_x^\varphi.M$ instead of $\mathsf{Y}(\lambda \varphi x.M)$ for the fixpoint construct. The branching construct is $\mathsf{if}(M, N, P)$, which evaluates to N if $M \leq 0$ and P otherwise. In SPCF, sample draws a random value from the uniform distribution on $[0, 1]$, and $\mathsf{score}(M)$ weights the current execution with the value of M. Samples from a different real-valued distribution D can be obtained by applying the inverse of the cumulative density function for D to a uniform sample [52]. Most PPLs feature an **observe** statement instead of manipulating the likelihood weight directly with score, but they are equally expressive [57].[5] As usual, we write $\mathsf{let}\, x = M \,\mathsf{in}\, N$ for $(\lambda x.N)M$, $M; N$ for $\mathsf{let}\, _ = M \,\mathsf{in}\, N$ and $M \oplus_p N$ for $\mathsf{if}(\mathsf{sample} - p, M, N)$. The type system of our language is as expected, with simple types being generated by $\alpha, \beta := \mathbf{R} \mid \alpha \to \beta$. Selected rules are given below:

$$\frac{}{\Gamma \vdash \mathsf{sample} : \mathbf{R}} \qquad \frac{\Gamma \vdash M : \mathbf{R}}{\Gamma \vdash \mathsf{score}(M) : \mathbf{R}}$$

$$\frac{\Gamma, \varphi : \alpha \to \beta, x : \alpha \vdash M : \beta}{\Gamma \vdash \mu_x^\varphi.M : \alpha \to \beta} \qquad \frac{\{\Gamma \vdash M_i : \mathbf{R}\}_{i=1}^{|f|}}{\Gamma \vdash \underline{f}(M_1, \cdots, M_{|f|}) : \mathbf{R}}$$

[5] In Bayesian terms, an **observe** statement multiplies the likelihood function by the probability (density) of the observation [32] (as we have used in Example 1.1). Scoring makes this explicit by keeping a weight for each program execution [8]. Observing a value v from a distribution D then simply multiplies the current weight by $\mathrm{pdf}_D(v)$ where pdf_D is the probability density function of D (for continuous distributions) or the probability mass function of D (for discrete distributions).

2.3 Trace Semantics

Following [8], we endow SPCF with a trace-based operational semantics. We evaluate a probabilistic program P on a fixed ***trace*** $\boldsymbol{s} = \langle r_1, \dots, r_n \rangle \in \mathbb{T} := \bigcup_{n\in\mathbb{N}}[0,1]^n$, which *predetermines* the probabilistic choices made during the evaluation. Our semantics therefore operates on configurations of the form $(M, \boldsymbol{s}, w)$ where M is an SPCF term, $\boldsymbol{s}$ is a trace and $w \in \mathbb{R}_{\geq 0}$ a weight. The call-by-value (CbV) reduction is given by the rules in Fig. 2, where $E[\cdot]$ denotes a CbV evaluation context. The definition is standard [4, 8, 39]. Given a program $\vdash P : \mathbf{R}$, we call a trace $\boldsymbol{s}$ ***terminating*** just if $(P, \boldsymbol{s}, 1) \to^* (V, \langle\rangle, w)$ for some value V and weight w, i.e. if the samples drawn are as specified by $\boldsymbol{s}$, the program P terminates. Note that we require the trace $\boldsymbol{s}$ to be completely used up. As P is of type $\mathbf{R}$ we can assume that $V = \underline{r}$ for some $r \in \mathbb{R}$. Each terminating trace $\boldsymbol{s}$ therefore uniquely determines the returned *value* $\underline{r}$ where $r =: \mathrm{val}_P(\boldsymbol{s}) \in \mathbb{R}$, and the *weight* $w =: \mathrm{wt}_P(\boldsymbol{s}) \in \mathbb{R}_{\geq 0}$, of the execution. For a nonterminating trace $\boldsymbol{s}$, $\mathrm{val}_P(\boldsymbol{s})$ is undefined and $\mathrm{wt}_P(\boldsymbol{s}) := 0$.

Example 2.1. Consider Example 1.1. On the trace $\boldsymbol{s} = \langle 0.1, 0.2, 0.4, 0.7, 0.8\rangle \in [0,1]^5 \subseteq \mathbb{T}$, the pedestrian walks 0.2 away from their home (taking the left branch of $\oplus_{0.5}$ as $0.4 \leq 0.5$) and 0.7 towards their home (as $0.8 > 0.5$), hence:

$$\mathrm{val}_P(\boldsymbol{s}) = 3 \times 0.1 = 0.3, \;\; \mathrm{wt}_P(\boldsymbol{s}) = \mathrm{pdf}_{\mathrm{Normal}(1.1,0.1)}(0.9).$$

In order to do measure theory, we need to turn our set of traces into a measurable space. The trace space $\mathbb{T}$ is equipped with the σ-algebra $\Sigma_{\mathbb{T}} := \{\bigcup_{n\in\mathbb{N}} U_n \mid U_n \in \Sigma_{[0,1]^n}\}$ where $\Sigma_{[0,1]^n}$ is the Borel σ-algebra on $[0,1]^n$. We define a measure $\mu_{\mathbb{T}}$ by $\mu_{\mathbb{T}}(U) := \sum_{n\in\mathbb{N}} \lambda_n(U \cap [0,1]^n)$, as in [8].

We can now define the semantics of an SPCF program $\vdash P : \mathbf{R}$ by using the weight and returned value of (executions of P determined by) individual traces. Given $U \in \Sigma_{\mathbb{R}}$, we need to define the likelihood of P evaluating to a value in U. To this end, we set $\mathrm{val}_P^{-1}(U) := \{\boldsymbol{s} \in \mathbb{T} \mid (P, \boldsymbol{s}, 1) \to^* (\underline{r}, \langle\rangle, w), r \in U\}$, i.e. the set of traces on which the program P reduces to a value in U. As shown in [8, Lem. 9], $\mathrm{val}_P^{-1}(U)$ is measurable. Thus, we can define (cf. [8, 39])

$$[\![P]\!](U) := \int_{\mathrm{val}_P^{-1}(U)} \mathrm{wt}_P(\boldsymbol{s})\, \mu_{\mathbb{T}}(\mathrm{d}\boldsymbol{s}).$$

That is, the integral takes all traces $\boldsymbol{s}$ on which P evaluates to a value in U, weighting each $\boldsymbol{s}$ with the weight $\mathrm{wt}_P(\boldsymbol{s})$ of the corresponding execution. A program P is called ***almost surely terminating (AST)*** if it terminates with probability 1, i.e. $\mu_{\mathbb{T}}(\mathrm{val}_P^{-1}(\mathbb{R})) = 1$. This is a necessary assumption for approximate inference algorithms (since they execute the program). See [8] for a more in-depth discussion of this (standard) sampling-style semantics.

Normalizing constant and integrability. In Bayesian statistics, one is usually interested in the *normalised* posterior, which is a conditional probability distribution. We obtain the normalised denotation as $\mathrm{posterior}_P := \frac{[\![P]\!]}{Z_P}$ where $Z_P := [\![P]\!](\mathbb{R})$ is the ***normalising constant***. We call P ***integrable*** if $0 < Z_P < \infty$. The bounds computed in this paper (on the unnormalised denotation $[\![P]\!]$) allow us to compute bounds on the normalizing constant Z_P, and thereby also on the normalised denotation. All bounds reported in this paper (in particular in Section 7) refer to the *normalised* denotation.

3 Interval Trace Semantics

In order to obtain guaranteed bounds on the distribution denotation $[\![P]\!]$ (and also on $\mathrm{posterior}_P$) of a program P, we present an interval-based semantics. In our semantics, we approximate the outcomes of sample with intervals and handle arithmetic operations by means of interval arithmetic (which is similar to the approach by Beutner and Ong [4] in the context of termination analysis). Our semantics enables us to reason about the denotation of a program *without* considering the uncountable space of traces explicitly.

3.1 Interval Arithmetic

For our purposes, an ***interval*** has the form $[a, b]$ which denotes the set $\{x \in \mathbb{R} \mid a \leq x \leq b\}$, where $a \in \mathbb{R} \cup \{-\infty\}$, $b \in \mathbb{R} \cup \{\infty\}$, and $a \leq b$. For consistency, we write $[0, \infty]$ instead of the more typical $[0, \infty)$. For $X \subseteq \mathbb{R} \cup \{-\infty, \infty\}$, we denote by $\mathbb{I}_X$ the set of intervals with endpoints in X, and simply write $\mathbb{I}$ for $\mathbb{I}_{\mathbb{R}\cup\{-\infty,\infty\}}$. An n-dimensional ***box*** is the Cartesian product of n intervals.

We can lift functions on real numbers to intervals as follows: for each $f : \mathbb{R}^n \to \mathbb{R}$ we define $f^{\mathbb{I}} : \mathbb{I}^n \to \mathbb{I}$ by

$$f^{\mathbb{I}}([a_1, b_1], \dots, [a_n, b_n]) := [\inf F, \sup F]$$

where $F := f([a_1, b_1], \dots, [a_n, b_n])$. For common functions like $+, -, \times, |\cdot|$, min, max, and monotonically increasing or decreasing functions $f : \mathbb{R} \to \mathbb{R}$, their interval-lifted counterparts can easily be computed, from the values of the original function on just the endpoints of the input interval. For example, addition lifts to $[a_1, b_1] +^{\mathbb{I}} [a_2, b_2] = [a_1 + a_2, b_1 + b_2]$; similarly for multiplication $\times^{\mathbb{I}}$.

3.2 Interval Traces and Interval SPCF

In our interval interpretation, probabilistic programs are run on *interval traces*. An ***interval trace***, $\langle I_1, \dots, I_n \rangle \in \mathbb{T}_{\mathbb{I}} := \bigcup_{n\in\mathbb{N}} (\mathbb{I}_{[0,1]})^n$, is a finite sequence of intervals $I_1, \dots, I_n$, each with endpoints between 0 and 1. To distinguish ordinary traces $\boldsymbol{s} \in \mathbb{T}$ from interval traces $\boldsymbol{t} \in \mathbb{T}_{\mathbb{I}}$, we call the former *concrete* traces.

We define the ***refinement*** relation $\triangleleft$ between concrete and interval traces as follows: for $\boldsymbol{s} = \langle r_1, \cdots, r_n \rangle \in \mathbb{T}$ and $\boldsymbol{t} = \langle I_1, \dots, I_m \rangle \in \mathbb{T}_{\mathbb{I}}$, we define $\boldsymbol{s} \triangleleft \boldsymbol{t}$ just if $n = m$ and for all i, $r_i \in I_i$. For each interval trace $\boldsymbol{t}$, we denote by $(\!|\boldsymbol{t}|\!) := \{\boldsymbol{s} \in \mathbb{T} \mid \boldsymbol{s} \triangleleft \boldsymbol{t}\}$ the set of all refinements of $\boldsymbol{t}$.

To define a reduction of a term on an interval trace, we extend SPCF with *interval literals* $\underline{[a, b]}$, which replace the literals $\underline{r}$ but are still considered values of type $\mathbf{R}$. In fact, $\underline{r}$

$$\frac{}{((\lambda x.M)V, \boldsymbol{t}, w) \to_{\mathbb{I}} (M[V/x], \boldsymbol{t}, w)} \qquad \frac{}{(\mathsf{sample}, I\,\boldsymbol{t}, w) \to_{\mathbb{I}} (\underline{I}, \boldsymbol{t}, w)}$$

$$\frac{}{((\mu^{\varphi}_{x}.M)V, \boldsymbol{t}, w) \to_{\mathbb{I}} (M[V/x, (\mu^{\varphi}_{x}.M)/\varphi], \boldsymbol{t}, w)}$$

$$\frac{b \leq 0}{(\mathsf{if}(\underline{[a,b]}, N, P), \boldsymbol{t}, w) \to_{\mathbb{I}} (N, \boldsymbol{t}, w)}$$

$$\frac{a > 0}{(\mathsf{if}(\underline{[a,b]}, N, P), \boldsymbol{t}, w) \to_{\mathbb{I}} (P, \boldsymbol{t}, w)}$$

$$\frac{}{(f(\underline{I_1}, \cdots, \underline{I_{|f|}}), \boldsymbol{t}, w) \to_{\mathbb{I}} (\underline{f^{\mathbb{I}}(I_1, \ldots, I_{|f|})}, \boldsymbol{t}, w)}$$

$$\frac{a \geq 0}{(\mathsf{score}(\underline{[a,b]}), \boldsymbol{t}, w) \to_{\mathbb{I}} (\underline{[a,b]}, \boldsymbol{t}, w \times^{\mathbb{I}} [a,b])}$$

$$\frac{(R, \boldsymbol{t}, w) \to_{\mathbb{I}} (M, \boldsymbol{t}', w')}{(E[R], \boldsymbol{t}, w) \to_{\mathbb{I}} (E[M], \boldsymbol{t}', w')}$$

Figure 3. Interval reduction rules for (interval) SPCF ($\to_{\mathbb{I}}$).

can be read as an abbreviation for $\underline{[r,r]}$. We call such terms ***interval terms***, and the resulting language ***Interval SPCF***.

Reduction. The interval-based reduction $\to_{\mathbb{I}}$ now operates on configurations $(M, \boldsymbol{t}, w)$ of interval terms M, interval traces $\boldsymbol{t} \in \mathbb{T}_{\mathbb{I}}$, and interval weights $w \in \mathbb{I}_{\mathbb{R}_{\geq 0} \cup \{\infty\}}$. The redexes and evaluation contexts of SPCF extend naturally to interval terms. The reduction rules are given in Fig. 3.[6]

Given a program $\vdash P : \mathbf{R}$, the reduction relation $\to_{\mathbb{I}}$ allows us to define the *interval weight* function ($\mathsf{wt}^{\mathbb{I}}_P : \mathbb{T}_{\mathbb{I}} \to \mathbb{I}_{\mathbb{R}_{\geq 0} \cup \{\infty\}}$) and *interval value* function ($\mathsf{val}^{\mathbb{I}}_P : \mathbb{T}_{\mathbb{I}} \to \mathbb{I}$) by:

$$\mathsf{wt}^{\mathbb{I}}_P(\boldsymbol{t}) := \begin{cases} w & \text{if } (P, \boldsymbol{t}, 1) \to^*_{\mathbb{I}} (\underline{I}, \langle\rangle, w) \\ [0, \infty] & \text{otherwise,} \end{cases}$$

$$\mathsf{val}^{\mathbb{I}}_P(\boldsymbol{t}) := \begin{cases} I & \text{if } (P, \boldsymbol{t}, 1) \to^*_{\mathbb{I}} (\underline{I}, \langle\rangle, w) \\ [-\infty, \infty] & \text{otherwise.} \end{cases}$$

It is not difficult to prove the following relationship between standard and interval reduction.

Lemma 3.1. *Let $\vdash P : \boldsymbol{R}$ be a program. For any interval trace $\boldsymbol{t}$ and concrete trace $\boldsymbol{s} \triangleleft \boldsymbol{t}$, we have $\mathsf{wt}_P(\boldsymbol{s}) \in \mathsf{wt}^{\mathbb{I}}_P(\boldsymbol{t})$ and $\mathsf{val}_P(\boldsymbol{s}) \in \mathsf{val}^{\mathbb{I}}_P(\boldsymbol{t})$ (provided $\mathsf{val}_P(\boldsymbol{s})$ is defined).*

3.3 Bounds from Interval Traces

Lower bounds. How can we use this interval trace semantics to obtain lower bounds on $[\![P]\!]$? We need a few definitions. Two intervals $[a_1, b_1], [a_2, b_2] \in \mathbb{I}$ are called ***almost disjoint*** if $b_1 \leq a_2$ or $b_2 \leq a_1$. Interval traces $\langle I_1, \ldots, I_m\rangle$ and $\langle J_1, \ldots, J_n\rangle \in \mathbb{T}_{\mathbb{I}}$ are called ***compatible*** if there is an index $i \in \{1, \ldots, \min(m,n)\}$ such that I_i and J_i are almost disjoint.

[6]For conditionals, the interval bound is not always precise enough to decide which branch to take, so the reduction can get stuck if $a \leq 0 < b$. We could include additional rules to overapproximate the branching behaviour (see the full version [5]). But the rules given here simplify the presentation and are sufficient to prove soundness and completeness.

A set of interval traces is called *compatible* if its elements are pairwise compatible. We define the ***volume*** of an interval trace $\boldsymbol{t} = \langle [a_1, b_1], \ldots, [a_n, b_n]\rangle$ as $\mathrm{vol}(\boldsymbol{t}) := \prod_{i=1}^{n}(b_i - a_i)$.

Let $\mathcal{T} \subseteq \mathbb{T}_{\mathbb{I}}$ be a countable and compatible set of interval traces. Define the *lower bound* on $[\![P]\!]$ by

$$\mathsf{lowerBd}^{\mathcal{T}}_P(U) := \sum_{\boldsymbol{t} \in \mathcal{T}} \mathrm{vol}(\boldsymbol{t}) \cdot (\min \mathsf{wt}^{\mathbb{I}}_P(\boldsymbol{t})) \cdot \left[\mathsf{val}^{\mathbb{I}}_P(\boldsymbol{t}) \subseteq U\right]$$

for $U \in \Sigma_{\mathbb{R}}$. That is, we sum over each interval trace in $\mathcal{T}$ whose value is *guaranteed* to be in U, weighted by its volume and the lower bound of its weight interval. Note that, in general, $\mathsf{lowerBd}^{\mathcal{T}}_P$ is not a measure, but merely a *superadditive measure*.[7]

Upper bounds. For upper bounds, we require the notion of a set of interval traces being *exhaustive*, which is easiest to express in terms of infinite traces. Let $\mathbb{T}_\infty := [0,1]^\omega$ be the set of infinite traces. Every interval trace $\boldsymbol{t}$ *covers* the set of infinite traces with a prefix contained in $\boldsymbol{t}$, i.e. $\mathit{cover}(\boldsymbol{t}) := (\!|\boldsymbol{t}|\!) \times \mathbb{T}_\infty$ (where the Cartesian product $\times$ can be viewed as trace concatenation). A countable set of (finite) interval traces $\mathcal{T} \subseteq \mathbb{T}_{\mathbb{I}}$ is called ***exhaustive*** if $\bigcup_{\boldsymbol{t} \in \mathcal{T}} \mathit{cover}(\boldsymbol{t})$ covers almost all of $\mathbb{T}_\infty$, i.e. $\mu_{\mathbb{T}_\infty}(\mathbb{T}_\infty \setminus \bigcup_{\boldsymbol{t} \in \mathcal{T}} \mathit{cover}(\boldsymbol{t})) = 0$.[8] Phrased differently, almost all concrete traces must have a finite prefix that is contained in some interval trace in $\mathcal{T}$. Therefore, the analysis in the interval semantics on $\mathcal{T}$ covers the behaviour on almost all concrete traces (in the original semantics).

Example 3.1. (i) The singleton set $\{\langle [0,1], [0, 0.6]\rangle\}$ is not exhaustive as, e.g. all infinite traces $\langle r_1, r_2, \cdots\rangle$ with $r_2 > 0.6$ are not covered. (ii) The set $\{\langle [0, 0.6]\rangle, \langle [0.3, 1]\rangle\}$ is exhaustive, but not compatible. (iii) Define $\mathcal{T}_1 := \{\langle [\frac{1}{2}, 1]^{\cdots n}, [0, \frac{1}{3}]\rangle \mid n \in \mathbb{N}\}$ and $\mathcal{T}_2 := \{\langle [\frac{1}{2}, 1]^{\cdots n}, [0, \frac{1}{2}]\rangle \mid n \in \mathbb{N}\}$ where $x^{\cdots n}$ denotes n-fold repetition of x. $\mathcal{T}_1$ is compatible but not exhaustive. For example, it doesn't cover the set $[\frac{1}{2}, 1] \times (\frac{1}{3}, \frac{1}{2}) \times \mathbb{T}_\infty$, i.e. all traces $\langle r_1, r_2, \cdots\rangle$ where $r_1 \in [\frac{1}{2}, 1]$ and $r_2 \in (\frac{1}{3}, \frac{1}{2})$. $\mathcal{T}_2$ is compatible and exhaustive (the set of non-covered traces $(\frac{1}{2}, 1]^\omega$ has measure 0).

Let $\mathcal{T} \subseteq \mathbb{T}_{\mathbb{I}}$ be a countable and exhaustive set of interval traces. Define the *upper bound* on $[\![P]\!]$ by

$$\mathsf{upperBd}^{\mathcal{T}}_P(U) := \sum_{\boldsymbol{t} \in \mathcal{T}} \mathrm{vol}(\boldsymbol{t}) \cdot (\sup \mathsf{wt}^{\mathbb{I}}_P(\boldsymbol{t})) \cdot \left[\mathsf{val}^{\mathbb{I}}_P(\boldsymbol{t}) \cap U \neq \emptyset\right]$$

for $U \in \Sigma_{\mathbb{R}}$. That is, we sum over each interval trace in $\mathcal{T}$ whose value *may* be in U, weighted by its volume and the upper bound of its weight interval. Note that $\mathsf{upperBd}^{\mathcal{T}}_P$ is not a measure but only a *subadditive measure*.[9]

[7]A *superadditive measure* μ on (Ω, Σ_Ω) is a measure, except that σ-additivity is replaced by σ-superadditivity: $\mu(\bigcup_{i \in \mathbb{N}} U_i) \geq \sum_{i \in \mathbb{N}} \mu(U_i)$ for a countable, pairwise disjoint family $(U_i)_{i \in \mathbb{N}} \in \Sigma_\Omega$.

[8]The σ-algebra on $\mathbb{T}_\infty$ is defined as the smallest σ-algebra that contains all sets $U \times \mathbb{T}_\infty$ where $U \in \Sigma_{[0,1]^n}$ for some $n \in \mathbb{N}$. The measure $\mu_{\mathbb{T}_\infty}$ is the unique measure with $\mu_{\mathbb{T}_\infty}(U \times \mathbb{T}_\infty) = \lambda_n(U)$ when $U \in \Sigma_{[0,1]^n}$.

[9]A *subadditive measure* μ on (Ω, Σ_Ω) is a measure, except that σ-additivity is replaced by σ-subadditivity: $\mu(\bigcup_{i \in \mathbb{N}} U_i) \leq \sum_{i \in \mathbb{N}} \mu(U_i)$ for a countable, pairwise disjoint family $(U_i)_{i \in \mathbb{N}} \in \Sigma_\Omega$.

4 Soundness and Completeness

4.1 Soundness

We show that the two bounds described above are *sound*, in the following sense.

Theorem 4.1 (Sound lower bounds). *Let $\mathcal{T}$ be a countable and compatible set of interval traces and $\vdash P : \boldsymbol{R}$ a program. Then* $\mathsf{lowerBd}_P^{\mathcal{T}} \leq [\![P]\!]$.

Proof. For any $U \in \Sigma_{\mathbb{R}}$, we have:

$$\begin{aligned}
\mathsf{lowerBd}_P^{\mathcal{T}}(U) &= \sum_{t \in \mathcal{T}} \mathrm{vol}(\boldsymbol{t})(\min \mathrm{wt}_P^{\mathbb{I}}(\boldsymbol{t}))\left[\mathrm{val}_P^{\mathbb{I}}(\boldsymbol{t}) \subseteq U\right] \\
&= \sum_{t \in \mathcal{T}} \int_{(\!|\boldsymbol{t}|\!)} (\min \mathrm{wt}_P^{\mathbb{I}}(\boldsymbol{t}))\left[\mathrm{val}_P^{\mathbb{I}}(\boldsymbol{t}) \subseteq U\right] \mathrm{d}\boldsymbol{s} \\
&\leq \sum_{t \in \mathcal{T}} \int_{(\!|\boldsymbol{t}|\!)} \mathrm{wt}_P(\boldsymbol{s})\left[\mathrm{val}_P(\boldsymbol{s}) \in U\right] \mathrm{d}\boldsymbol{s} \qquad (1) \\
&= \int_{\bigcup_{t \in \mathcal{T}} (\!|\boldsymbol{t}|\!)} \mathrm{wt}_P(\boldsymbol{s})\left[\mathrm{val}_P(\boldsymbol{s}) \in U\right] \mathrm{d}\boldsymbol{s} \qquad (2) \\
&\leq \int_{\mathbb{T}} \mathrm{wt}_P(\boldsymbol{s})\left[\mathrm{val}_P(\boldsymbol{s}) \in U\right] \mathrm{d}\boldsymbol{s} = [\![P]\!](U) \qquad (3)
\end{aligned}$$

where Eq. (1) follows from Lemma 3.1, Eq. (2) from compatibility, and Eq. (3) from $\bigcup_{t \in \mathcal{T}} (\!|\boldsymbol{t}|\!) \subseteq \mathbb{T}$. ◻

Theorem 4.2 (Sound upper bounds). *Let $\mathcal{T}$ be a countable and exhaustive set of interval traces and $\vdash P : \boldsymbol{R}$ a program. Then* $[\![P]\!] \leq \mathsf{upperBd}_P^{\mathcal{T}}$.

Proof sketch. The formal proof is similar to the soundness proof for the lower bound in Theorem 4.1, but needs an infinite trace semantics [16] for probabilistic programs and is given in the full version [5]. The idea is that each interval trace $\boldsymbol{t}$ summarises all infinite traces starting with $(\!|\boldsymbol{t}|\!)$, i.e. all traces in $\mathit{cover}(\boldsymbol{t})$. Exhaustivity ensures that almost all infinite traces are "covered". ◻

4.2 Completeness

The soundness results for upper and lower bounds allow us to derive bounds on the denotation of a program. One would expect that a finer partition of interval traces will yield more precise bounds. In this section, we show that for a program P and an interval $I \in \mathbb{I}$, the approximations $\mathsf{lowerBd}_P^{\mathcal{T}}(I)$ and $\mathsf{upperBd}_P^{\mathcal{T}}(I)$ can in fact come arbitrarily close to $[\![P]\!](I)$ for suitable $\mathcal{T}$. However, this is only possible under certain assumptions.

Assumption 1: use of sampled values. Interval arithmetic is imprecise if the same value is used more than once: consider, for instance, $\mathsf{let}\, s = \mathsf{sample}\, \mathsf{in}\, \mathsf{if}(s - s, 0, 1)$ which deterministically evaluates to 0. However, in interval arithmetic, if x is approximated by an interval $[a, b]$ with $a < b$, the difference $x - x$ is approximated as $[a - b, b - a]$, which always contains both positive and negative values. So no non-trivial interval trace can separate the two branches.

To avoid this, we could consider a call-by-name semantics (as done in [4]) where sample values can only be used once by definition. However, many of our examples cannot be expressed in the call-by-name setting, so we instead propose a less restrictive criterion to guarantee completeness for call-by-value: we allow sample values to be used more than once, but at most once in the guard of each conditional, at most once in each score expression, and at most once in the return value. While this prohibits terms like the one above, it allows, e.g. $\mathsf{let}\, s = \mathsf{sample}\, \mathsf{in}\, \mathsf{if}(s, \underline{f}(s), \underline{g}(s))$. This sufficient condition is formalised in the full version [5]. Most examples we encountered in the literature satisfy this assumption.

Assumption 2: primitive functions. In addition, we require mild assumptions on the primitive functions, called *boxwise continuity* and *interval separability*.

We need to be able to approximate a program's weight function by step functions in order to obtain tight bounds on its integral. A function $f : \mathbb{R}^n \to \mathbb{R}$ is ***boxwise continuous*** if it can be written as the countable union of continuous functions on boxes, i.e. if there is a countable union of pairwise almost disjoint boxes B_i such that $\bigcup B_i = \mathbb{R}^n$ and the restriction $f|_{B_i}$ is continuous for each B_i.

Furthermore, we need to approximate preimages. Formally, we say that A is a ***tight subset*** of B (written $A \Subset B$) if $A \subseteq B$ and $B \setminus A$ is a null set. A function $f : \mathbb{R}^n \to \mathbb{R}$ is called ***interval separable*** if for every interval $[a, b] \in \mathbb{I}$, there is a countable set $\mathcal{B}$ of boxes in $\mathbb{R}^n$ that tightly approximates the preimage, i.e. $\bigcup \mathcal{B} \Subset f^{-1}([a, b])$. A sufficient condition for checking this is the following. If f is boxwise continuous and preimages of points have measure zero, then f is already interval separable (cf. the full version [5]).

We assume the set $\mathcal{F}$ of primitive functions is closed under composition and each $f \in \mathcal{F}$ is boxwise continuous and interval separable.

The completeness theorem. Using these two assumptions, we can state completeness of our interval semantics.

Theorem 4.3 (Completeness of interval approximations). *Let $I \in \mathbb{I}$ and $\vdash P : \boldsymbol{R}$ be an almost surely terminating program satisfying the two assumptions discussed above. Then, for all $\epsilon > 0$, there is a countable set of interval traces $\mathcal{T} \subseteq \mathbb{T}_{\mathbb{I}}$ that is compatible and exhaustive such that*

$$\mathsf{upperBd}_P^{\mathcal{T}}(I) - \epsilon \leq [\![P]\!](I) \leq \mathsf{lowerBd}_P^{\mathcal{T}}(I) + \epsilon.$$

Proof sketch. We consider each branching path through the program separately. The set of relevant traces for a given path is a preimage of intervals under compositions of interval separable functions, hence can essentially be partitioned into boxes. By boxwise continuity, we can refine this partition such that the weight function is continuous on each box. To approximate the integral, we pass to a refined partition again, essentially computing Riemann sums. The latter converge

to the Riemann integral, which agrees with the Lebesgue integral under our conditions, as desired. □

For the lower bound, we can actually derive ϵ-close bounds using only finitely many interval traces:

Corollary 4.4. *Let $I \in \mathbb{I}$ and $\vdash P : \boldsymbol{R}$ be as in Theorem 4.3. There is a sequence of finite, compatible sets of interval traces $\mathcal{T}_1, \mathcal{T}_2, \ldots \subseteq \mathbb{T}_{\mathbb{I}}$ s.t.* $\lim_{n\to\infty} \mathsf{lowerBd}_P^{\mathcal{T}_n}(I) = [\![P]\!](I)$.

For the upper bound, a restriction to finite sets $\mathcal{T}$ of interval traces is, in general, not possible: if the weight function for a program is unbounded, it is also unbounded on some $\boldsymbol{t} \in \mathcal{T}$. Then $\mathsf{wt}_P^{\mathbb{I}}(\boldsymbol{t})$ is an infinite interval, implying $\mathsf{upperBd}_P^{\mathcal{T}}(I) = \infty$ (see the full version [5] for details). Despite the (theoretical) need for countably infinite many interval traces, we can, in many cases, compute finite upper bounds by making use of an interval-based static approximation, formalised as a type system in the next section.

5 Weight-aware Interval Type System

To obtain sound bounds on the denotation with only finitely many interval traces, we present an interval-based type system that can derive static bounds on a program. Crucially, our type-system is *weight-aware*: we bound not only the return value of a program but also the weight of an execution. Our analyzer GuBPI uses it for two purposes. First, it allows us to derive upper bounds even for areas of the sample space not covered with interval traces. Second, we can use our analysis to derive a *finite* (and sound) approximation of the infinite number of symbolic execution paths of a program (more details are given in Section 6). Note that the bounds inferred by our system are *interval bounds*, which allow for seamless integration with our interval trace semantics. In this section, we present the interval type system and sketch a constraint-based type inference method.

5.1 Interval Types

We define interval types by the following grammar:

$$\sigma := I \mid \sigma \to \mathcal{A} \qquad \mathcal{A} := \begin{Bmatrix} \sigma \\ I \end{Bmatrix}$$

where $I \in \mathbb{I}$ is an interval. For readers familiar with refinement types, it is easiest to view the type $\sigma = I$ as the refinement type $\{x : \mathbb{R} \mid x \in I\}$. The definition of the syntactic category $\mathcal{A}$ by mutual recursion with σ gives a bound on the weight of the execution. We call a type σ *weightless* and a type $\mathcal{A}$ *weighted*. The following examples should give some intuition about the types.

Example 5.1. Consider the example term

$$\big(\mu_x^{\varphi}.\, 5 \times x \oplus_{0.5} \mathit{sigm}(\varphi\, x + \mathsf{score}\, \mathsf{sample})\big)(4 \times \mathsf{sample})$$

where $\mathit{sigm} : \mathbb{R} \to [0,1]$ is the sigmoid function. In our type system, this term can be typed with the weighted type $\begin{Bmatrix} [0,20] \\ [0,1] \end{Bmatrix}$, which indicates that any terminating execution of the term reduces to a value (a number) within $[0, 20]$ and the weight of any such execution lies within $[0, 1]$.

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \begin{Bmatrix} \sigma \\ \mathbf{1} \end{Bmatrix}} \qquad \frac{\Gamma \vdash M : \mathcal{A} \quad \mathcal{A} \sqsubseteq_{\mathcal{A}} \mathcal{B}}{\Gamma \vdash M : \mathcal{B}} \qquad \frac{\Gamma; \varphi : \sigma \to \mathcal{A}; x : \sigma \vdash M : \mathcal{A}}{\Gamma \vdash \mu_x^{\varphi}.M : \begin{Bmatrix} \sigma \to \mathcal{A} \\ \mathbf{1} \end{Bmatrix}}$$

$$\frac{\Gamma; x : \sigma \vdash M : \mathcal{A}}{\Gamma \vdash \lambda x.M : \begin{Bmatrix} \sigma \to \mathcal{A} \\ \mathbf{1} \end{Bmatrix}} \qquad \frac{\Gamma \vdash M : \begin{Bmatrix} \sigma_1 \to \begin{Bmatrix} \sigma_2 \\ [e,f] \end{Bmatrix} \\ [a,b] \end{Bmatrix} \quad \Gamma \vdash N : \begin{Bmatrix} \sigma_1 \\ [c,d] \end{Bmatrix}}{\Gamma \vdash MN : \begin{Bmatrix} \sigma_2 \\ [a,b] \times^{\mathbb{I}} [c,d] \times^{\mathbb{I}} [e,f] \end{Bmatrix}}$$

$$\frac{}{\Gamma \vdash \underline{r} : \begin{Bmatrix} [r,r] \\ \mathbf{1} \end{Bmatrix}} \qquad \frac{\Gamma \vdash M : \begin{Bmatrix} [_,_] \\ [a,b] \end{Bmatrix} \quad \Gamma \vdash N : \begin{Bmatrix} \sigma \\ [c,d] \end{Bmatrix} \quad \Gamma \vdash P : \begin{Bmatrix} \sigma \\ [c,d] \end{Bmatrix}}{\Gamma \vdash \mathsf{if}(M,N,P) : \begin{Bmatrix} \sigma \\ [a,b] \times^{\mathbb{I}} [c,d] \end{Bmatrix}}$$

$$\frac{}{\Gamma \vdash \mathsf{sample} : \begin{Bmatrix} [0,1] \\ \mathbf{1} \end{Bmatrix}} \qquad \frac{\Gamma \vdash M : \begin{Bmatrix} [a,b] \\ [c,d] \end{Bmatrix}}{\Gamma \vdash \mathsf{score}(M) : \begin{Bmatrix} [a,b] \sqcap [0,\infty] \\ [c,d] \times^{\mathbb{I}} ([a,b] \sqcap [0,\infty]) \end{Bmatrix}}$$

$$\frac{\Gamma \vdash M_1 : \begin{Bmatrix} [a_1,b_1] \\ [c_1,d_1] \end{Bmatrix} \quad \cdots \quad \Gamma \vdash M_{|f|} : \begin{Bmatrix} [a_{|f|},b_{|f|}] \\ [c_{|f|},d_{|f|}] \end{Bmatrix}}{\Gamma \vdash f(M_1,\cdots,M_{|f|}) : \begin{Bmatrix} f^{\mathbb{I}}([a_1,b_1],\cdots,[a_{|f|},b_{|f|}]) \\ (\times^{\mathbb{I}})_{i=1}^{|f|}[c_i,d_i] \end{Bmatrix}}$$

Figure 4. Weight-aware interval type system for SPCF. We abbreviate $\mathbf{1} := [1, 1]$.

Example 5.2. We consider the fixpoint subexpression of the pedestrian example in Example 1.1 which is

$$\mu_x^{\varphi}.\mathsf{if}(x, 0, \big(\lambda step.step + \varphi((x{+}step) \oplus_{0.5} (x{-}step))\big)\mathsf{sample}).$$

Using the typing rules (defined below), we can infer the type $\begin{Bmatrix} [a,b] \to \begin{Bmatrix} [0,\infty] \\ [1,1] \end{Bmatrix} \\ [1,1] \end{Bmatrix}$ for any a, b. This type indicates that any terminating execution reduces to a function value (of simple type $\mathbf{R} \to \mathbf{R}$) with weight within $[1, 1]$. If this function value is then called on a value within $[a, b]$, any terminating execution reduces to a value within $[0, \infty]$ with a weight within $[1, 1]$.

Subtyping. The partial order on intervals naturally extends to our type system. For base types I_1 and I_2, we define $I_1 \sqsubseteq_\sigma I_2$ just if $I_1 \sqsubseteq I_2$, where $\sqsubseteq$ is interval inclusion. We then extend this via:

$$\frac{\sigma_2 \sqsubseteq_\sigma \sigma_1 \qquad \mathcal{A}_1 \sqsubseteq_{\mathcal{A}} \mathcal{A}_2}{\sigma_1 \to \mathcal{A}_1 \sqsubseteq_\sigma \sigma_2 \to \mathcal{A}_2} \qquad \frac{\sigma_1 \sqsubseteq_\sigma \sigma_2 \qquad I_1 \sqsubseteq I_2}{\begin{Bmatrix} \sigma_1 \\ I_1 \end{Bmatrix} \sqsubseteq_{\mathcal{A}} \begin{Bmatrix} \sigma_2 \\ I_2 \end{Bmatrix}}$$

Note that in the case of weighted types, the subtyping requires not only that the weightless types be subtype-related ($\sigma_1 \sqsubseteq_\sigma \sigma_2$) but also that the weight bound be refined $I_1 \sqsubseteq I_2$. It is easy to see that both $\sqsubseteq_{\mathcal{A}}$ and $\sqsubseteq_\sigma$ are partial orders on types with the same underlying base type.

5.2 Type System

As for the interval-based semantics, we assume that every primitive operation $f : \mathbb{R}^n \to \mathbb{R}$ has an overapproximating interval abstraction $f^{\mathbb{I}} : \mathbb{I}^n \to \mathbb{I}$ (cf. Section 3.1). Interval typing judgments have the form $\Gamma \vdash M : \mathcal{A}$ where Γ is a typing context mapping variables to types σ. They are given via the rules in Fig. 4. Our system is sound in the following sense (which we here only state for first-order programs).

Theorem 5.1. *Let $\vdash P : \boldsymbol{R}$ be a simply-typed program. If $\vdash P : \left\{\begin{smallmatrix}[a,b]\\ [c,d]\end{smallmatrix}\right\}$ and $(P, \boldsymbol{s}, 1) \to^* (\underline{r}, \langle\rangle, w)$ for some $\boldsymbol{s} \in \mathbb{T}$ and $r, w \in \mathbb{R}$, then $r \in [a,b]$ and $w \in [c,d]$.*

Note that the bounds derived by our type system only refer to terminating executions, i.e. they are partial correctness statements. Theorem 5.1 formalises the intuition of an interval type, i.e. every type derivation in our system bounds *both* the returned value (in typical refinement-type fashion [23]) and the weight of this derivation. Our type system also comes with a weak completeness statement: for each term, we can derive some bounds in our system.

Proposition 5.2. *Let $\vdash P : \alpha$ be a simply-typed program. There exists a weighted interval type $\mathcal{A}$ such that $\vdash P : \mathcal{A}$.*

5.3 Constraint-based Type Inference

In this section, we briefly discuss the automated type *inference* in our system, as needed in our tool GuBPI. For space reasons, we restrict ourselves to an informal overview (see the full version [5] for a full account).

Given a program P, we can derive the symbolic skeleton of a type derivation (the structure of which is determined by P), where each concrete interval is replaced by a placeholder variable. The validity of a typing judgment within this skeleton can then be encoded as constraints. Crucially, as we work in the fixed interval domain and the subtyping structure $\sqsubseteq_{\mathcal{A}}$ is compositional, they are simple constraints over the placeholder variables in the abstract interval domain. Solving the resulting constraints naïvely might not terminate since the interval abstract domain is not chain-complete. Instead, we approximate the least fixpoint (where the fixpoint denotes a solution to the constraints) using *widening*, a standard approach to ensure termination of static analysis on domains with infinite chains [14, 15]. This is computationally much cheaper compared to, say, types with general first-order refinements where constraints are typically phrased as constrained Horn clauses (see e.g. [12]). This gain in efficiency is crucial to making our GuBPI tool practical.

6 Symbolic Execution and GuBPI

In this section, we describe the overall structure of our tool GuBPI (gubpi-tool.github.io), which builds upon symbolic execution. We also outline how the interval-based semantics can be accelerated for programs containing linear subexpressions.

6.1 Symbolic Execution

The starting point of our analysis is a *symbolic exploration* of the term in question [11, 27, 39]. For space reasons we only give an informal overview of the approach. A detailed and formal discussion can be found in the full version [5].

The idea of symbolic execution is to treat outcomes of sample expressions fully symbolically: each sample evaluates to a fresh variable ($\alpha_1, \alpha_2, \dots$), called *sample variable*. The result of symbolic execution is thus a symbolic value: a term consisting of sample variables and delayed primitive function applications. We postpone branching decisions and the weighting with score expressions because the value in question is symbolic. During execution, we therefore explore both branches of a conditional and keep track of the (symbolic) conditions on the sample variables that need to hold in the current branch. Similarly, we record the (symbolic) values of score expressions. Formally, our symbolic execution operates on ***symbolic configurations*** of the form $\psi = (\mathcal{M}, n, \Delta, \Xi)$ where $\mathcal{M}$ is a symbolic term containing sample variables instead of sample outcomes; $n \in \mathbb{N}$ is a natural number used to obtain fresh sample variables; Δ is a list of symbolic constraints of the form $\mathcal{V} \bowtie r$, where $\mathcal{V}$ is a symbolic value, $r \in \mathbb{R}$ and $\bowtie \in \{\leq, <, >, \geq\}$, to keep track of the conditions for the current execution path; and Ξ is a set of values that records all symbolic values of score expressions encountered along the current path. The symbolic reduction relation $\rightsquigarrow$ includes the following key rules.

$$(\mathsf{sample}, n, \Delta, \Xi) \rightsquigarrow (\alpha_{n+1}, n+1, \Delta, \Xi)$$
$$(\mathsf{if}(\mathcal{V}, \mathcal{N}, \mathcal{P})), n, \Delta, \Xi) \rightsquigarrow (\mathcal{N}, n, \Delta \cup \{\mathcal{V} \leq 0\}, \Xi)$$
$$(\mathsf{if}(\mathcal{V}, \mathcal{N}, \mathcal{P})), n, \Delta, \Xi) \rightsquigarrow (\mathcal{P}, n, \Delta \cup \{\mathcal{V} > 0\}, \Xi)$$
$$(\mathsf{score}(\mathcal{V}), n, \Delta, \Xi) \rightsquigarrow (\mathcal{V}, n, \Delta \cup \{\mathcal{V} \geq 0\}, \Xi \cup \{\mathcal{V}\})$$

That is, we replace sample outcomes with fresh sample variables (first rule), explore both paths of a conditional (second and third rule), and record all score values (fourth rule).

Example 6.1. Consider the symbolic execution of Example 1.1 where the first step moves the pedestrian towards their home (taking the right branch of $\oplus_{0.5}$) and the second step moves away from their home (the left branch of $\oplus_{0.5}$). We reach a configuration $(\mathcal{M}, 5, \Delta, \Xi)$ where $\mathcal{M}$ is

$$\mathsf{score}\big(\,\mathrm{pdf}_{\mathrm{Normal}(1.1,0.1)}\,(\alpha_2{+}\alpha_4{+}(\mu_x^{\varphi}.\mathcal{N})(3\alpha_1{-}\alpha_2{+}\alpha_4))\big); 3\alpha_1.$$

Here α_1 is the initial sample for *start*; α_2, α_4 the two samples of *step*; and α_3, α_5 the samples involved in the $\oplus_{0.5}$ operator. The fixpoint $\mu_x^{\varphi}.\mathcal{N}$ is already given in Example 5.2, $\Xi = \emptyset$ and $\Delta = \{3\alpha_1 > 0, \alpha_3 > \frac{1}{2}, 3\alpha_1 - \alpha_2 > 0, \alpha_5 \leq \frac{1}{2}\}$.

For a symbolic value $\mathcal{V}$ using sample variables $\overline{\alpha} = \alpha_1, \dots, \alpha_n$ and $\boldsymbol{s} \in [0,1]^n$, we write $\mathcal{V}[\boldsymbol{s}/\overline{\alpha}] \in \mathbb{R}$ for the substitution of concrete values in $\boldsymbol{s}$ for the sample variables. Call a symbolic configuration of the form $\Psi = (\mathcal{V}, n, \Delta, \Xi)$ (i.e. a configuration that has reached a symbolic value $\mathcal{V}$) a ***symbolic path***. We write *symPaths*(ψ) for the (countable) set of

Algorithm 1 Symbolic Analysis in GuBPI.

1: **Input:** Program $\vdash P : \mathbf{R}$, depth limit $D \in \mathbb{N}$, and $I \in \mathbb{I}$
2: $\psi_{\text{init}} := (P, 0, \emptyset, \emptyset); S := \{(\psi_{\text{init}}, 0)\}; T := \emptyset$
3: **while** $\exists(\psi, \mathit{depth}) \in S$ **do**
4: **if** ψ has terminated **then**
5: $T := T \cup \{\psi\}; S := S \setminus \{(\psi, \mathit{depth})\}$
6: **else if** ψ contains no fixpoints or $\mathit{depth} \leq D$ **then**
7: $S := S \setminus \{(\psi, \mathit{depth})\}$
8: **for** ψ' with $\psi \rightsquigarrow \psi'$ **do**
9: $S := S \cup \{(\psi', \mathit{depth} + 1)\}$
10: **else**
11: $S := (S \setminus \{(\psi, \mathit{depth})\}) \cup \{(\mathit{approxFix}(\psi), \mathit{depth})\}$
12: **return** $\big[\sum_{\Psi \in T} [\![\Psi]\!]_{lb}(I), \sum_{\Psi \in T} [\![\Psi]\!]_{ub}(I)\big]$

symbolic paths reached when evaluating from configuration ψ. Given a symbolic path $\Psi = (\mathcal{V}, n, \Delta, \Xi)$ and a set $U \in \Sigma_{\mathbb{R}}$, we define the denotation along Ψ, written $[\![\Psi]\!](U)$, as

$$\int_{[0,1]^n} [\mathcal{V}[s/\overline{\alpha}] \in U] \prod_{\mathcal{C} \bowtie r \in \Delta} [\mathcal{C}[s/\overline{\alpha}] \bowtie r] \prod_{\mathcal{W} \in \Xi} \mathcal{W}[s/\overline{\alpha}] \, \mathrm{d}s,$$

i.e. the integral of the product of the score weights Ξ over the traces of length n where the result value is in U and all the constraints Δ are satisfied. We can recover the denotation of a program P (as defined in Section 2) from all its symbolic paths starting from the configuration $(P, 0, \emptyset, \emptyset)$.

Theorem 6.1. *Let* $\vdash P : \boldsymbol{R}$ *be a program and* $U \in \Sigma_{\mathbb{R}}$*. Then*

$$[\![P]\!](U) = \textstyle\sum_{\Psi \in \mathit{symPaths}(P,0,\emptyset,\emptyset)} [\![\Psi]\!](U).$$

Analogously to interval SPCF (Section 3), we define ***symbolic interval terms*** as symbolic terms that may contain intervals (and similarly for symbolic interval values, symbolic interval configurations, and symbolic interval paths).

6.2 GuBPI

With symbolic execution at hand, we can outline the structure of our analysis tool GuBPI (sketched in Algorithm 1). GuBPI's analysis begins with symbolic execution of the input term to accumulate a set of symbolic *interval* paths T. If a symbolic configuration ψ has exceeded the user-defined depth limit D and still contains a fixpoint, we overapproximate all paths that extend ψ to ensure a finite set T. We accomplish this by using the interval type system (Section 5) to overapproximate all fixpoint subexpressions, thereby obtaining strongly normalizing terms (in line 11). Formally, given a symbolic configuration $\psi = (\mathcal{M}, n, \Delta, \Xi)$ we derive a typing judgment for the term $\mathcal{M}$ in the system from Section 5. Each first-order fixpoint subterm is thus given a (weightless) type of the form $[a,b] \to \left\{\begin{smallmatrix}[c,d]\\ [e,f]\end{smallmatrix}\right\}$. We replace this fixpoint with $\lambda_.\big(\mathrm{score}([e,f])\ ;\ [c,d]\big)$. We denote this operation on configurations by $\mathit{approxFix}(\psi)$ (it extends to higher-order fixpoints as expected). Note that $\mathit{approxFix}(\psi)$ is a symbolic *interval* configuration.

Example 6.2. Consider the symbolic configuration given in Example 6.1. As in Example 5.2 we infer the type of $\mu_x^{\varphi}.\mathcal{N}$ to be $[-1,4] \to \left\{\begin{smallmatrix}[0,\infty]\\ [1,1]\end{smallmatrix}\right\}$. The function *approxFix* replaces $\mu_x^{\varphi}.\mathcal{N}$ with $\lambda_.\mathrm{score}([1,1]); [0,\infty]$. By evaluating the resulting symbolic interval configuration further, we obtain the symbolic interval path $(3\alpha_1, 5, \Delta, \Xi)$ where Δ is as in Example 6.1 and $\Xi = \{\mathrm{pdf}_{\mathrm{Normal}(1.1,0.1)}(\alpha_2 + \alpha_4 + [0,\infty])\}$. Note that, in general, the further evaluation of $\mathit{approxFix}(\psi)$ can result in multiple symbolic interval paths.

Afterwards, we're left with a finite set T of symbolic interval paths. Due to the presence of intervals, we cannot define a denotation of such paths directly and instead define lower and upper bounds. For a symbolic interval value $\mathcal{V}$ that contains *no* sample variables, we define $\ulcorner\mathcal{V}\urcorner \subseteq \mathbb{R}$ as the set of all values that the term can evaluate to by replacing every interval $[a,b]$ with some value $r \in [a,b]$. Given a symbolic interval path $\Psi = (\mathcal{V}, n, \Delta, \Xi)$ and $U \in \Sigma_{\mathbb{R}}$ we define $[\![\Psi]\!]_{lb}(U)$ by considering only those concrete traces that fulfill the constraints in Ψ for *all* concrete values in the intervals and take the infimum over all scoring expressions:

$$\int [\ulcorner\mathcal{V}[s/\overline{\alpha}]\urcorner \subseteq U] \prod_{\mathcal{C} \bowtie r \in \Delta} [\forall t \in \ulcorner\mathcal{C}[s/\overline{\alpha}]\urcorner . t \bowtie r] \prod_{\mathcal{W} \in \Xi} \inf \ulcorner\mathcal{W}[s/\overline{\alpha}]\urcorner \, \mathrm{d}s.$$

Similarly, we define $[\![\Psi]\!]_{ub}(U)$ as

$$\int [\ulcorner\mathcal{V}[s/\overline{\alpha}]\urcorner \cap U \neq \emptyset] \prod_{\mathcal{C} \bowtie r \in \Delta} [\exists t \in \ulcorner\mathcal{C}[s/\overline{\alpha}]\urcorner . t \bowtie r] \prod_{\mathcal{W} \in \Xi} \sup \ulcorner\mathcal{W}[s/\overline{\alpha}]\urcorner \, \mathrm{d}s.$$

We note that, if Ψ contains no intervals, $[\![\Psi]\!]$ is defined and we have $[\![\Psi]\!]_{lb} = [\![\Psi]\!]_{ub} = [\![\Psi]\!]$. We can now state the following theorem that formalises the observation that $\mathit{approxFix}(\psi)$ soundly approximates all symbolic paths that result from ψ.

Theorem 6.2. *Let* ψ *be a symbolic (interval-free) configuration and* $U \in \Sigma_{\mathbb{R}}$*. Define* $A = \mathit{symPaths}(\psi)$ *as the (possibly infinite) set of all symbolic paths reached when evaluating* ψ *and* $B = \mathit{symPaths}(\mathit{approxFix}(\psi))$ *as the (finite) set of symbolic* interval *paths reached when evaluating* $\mathit{approxFix}(\psi)$*. Then*

$$\textstyle\sum_{\Psi \in B} [\![\Psi]\!]_{lb}(U) \leq \sum_{\Psi \in A} [\![\Psi]\!](U) \leq \sum_{\Psi \in B} [\![\Psi]\!]_{ub}(U).$$

The correctness of Algorithm 1 is then a direct consequence of Theorems 6.1 and 6.2:

Corollary 6.3. *Let* T *be the set of symbolic interval paths computed when at line 12 of Algorithm 1 and* $U \in \Sigma_{\mathbb{R}}$*. Then*

$$\textstyle\sum_{\Psi \in T} [\![\Psi]\!]_{lb}(U) \leq [\![P]\!](U) \leq \sum_{\Psi \in T} [\![\Psi]\!]_{ub}(U).$$

What remains is to compute (lower bounds on) $[\![\Psi]\!]_{lb}(I)$ and (upper bounds on) $[\![\Psi]\!]_{ub}(I)$ for a symbolic interval path $\Psi \in T$ and $I \in \mathbb{I}$. We first present the standard interval trace semantics (Section 6.3) and then a more efficient analysis for the case that Ψ contains only linear functions (Section 6.4).

6.3 Standard Interval Trace Semantics

For any symbolic interval path Ψ, we can employ the semantics as introduced in Section 3. Instead of applying the

analysis to the entire program, we can restrict to the current path Ψ (intuitively, by adding a score(0) to all other program paths). The interval traces split the domain of each sample variable in Ψ into intervals. It is easy to see that for any compatible and exhaustive set of interval traces $\mathcal{T}$, we have $\mathsf{lowerBd}^{\mathcal{T}}_{\Psi}(U) \leq [\![\Psi]\!]_{lb}(U)$ and $[\![\Psi]\!]_{ub}(U) \leq \mathsf{upperBd}^{\mathcal{T}}_{\Psi}(U)$ for any $U \in \Sigma_{\mathbb{R}}$ (see Theorem 4.1 and 4.2). Applying the interval-based semantics on the level of symbolic interval paths maintains the attractive features, namely soundness and completeness (relative to the current path) of the semantics. Note that the intervals occurring in Ψ seamlessly integrate with our interval-based semantics.

6.4 Linear Interval Trace Semantics

In case the score values and the guards of all conditionals are linear, we can improve and speed up the interval-based semantics.

Assume all symbolic interval values appearing in Ψ are interval linear functions of $\overline{\alpha}$ (i.e. functions $\overline{\alpha} \mapsto \mathbf{w}^\intercal \overline{\alpha} +^{\mathbb{I}} [a, b]$ for some $\mathbf{w} \in \mathbb{R}^n$ and $[a, b] \in \mathbb{I}$). We assume, for now, that each symbolic value $\mathcal{W} \in \Xi$ denotes an interval-free linear function (i.e. a function $\overline{\alpha} \mapsto \mathbf{w}^\intercal \overline{\alpha} + r$). Fix some interval $I \in \mathbb{I}$. We first note that both $[\![\Psi]\!]_{lb}(I)$ and $[\![\Psi]\!]_{ub}(I)$ are the integral of a polynomial over a convex polytope: define

$$\mathfrak{P}_{lb} := \Big\{ \mathbf{s} \in \mathbb{R}^n \mid \ulcorner \mathcal{V}[\mathbf{s}/\overline{\alpha}] \urcorner \subseteq I \wedge \bigwedge_{C \bowtie r \in \Delta} \forall t \in \ulcorner C[\mathbf{s}/\overline{\alpha}] \urcorner . t \bowtie r \Big\}$$

which is a polytope.[10] Then $[\![\Psi]\!]_{lb}(I)$ is the integral of the polynomial $\overline{\alpha} \mapsto \prod_{\mathcal{W} \in \Xi} \mathcal{W}$ over $\mathfrak{P}_{lb}$. The definition of $\mathfrak{P}_{ub}$ (as the region of integration for $[\![\Psi]\!]_{ub}(I)$) is similar. Such integrals can be computed exactly [2], e.g. with the LattE tool [20]. Unfortunately, our experiments showed that this does not scale to interesting probabilistic programs.

Instead, we derive guaranteed bounds on the denotation by means of iterated volume computations. This has the additional benefit that we can handle non-uniform samples and non-liner expressions in Ξ. We follow an approach similar to that of the interval-based semantics in Section 4 but do not split/bound *individual sample variables* and instead directly bound *linear functions* over the sample variables. Let $\Xi = \{\mathcal{W}_1, \cdots, \mathcal{W}_k\}$. We define a ***box*** (by abuse of language) as an element $\boldsymbol{t} = \langle [a_1, b_1], \cdots, [a_k, b_k] \rangle$, where $[a_i, b_i]$ gives a bound on $\mathcal{W}_i$.[11] We define $lb(\boldsymbol{t}) := \prod_{i=1}^k a_i$ and $ub(\boldsymbol{t}) := \prod_{i=1}^k b_i$. The box $\boldsymbol{t}$ naturally defines a subset of $\mathfrak{P}_{lb}$ given by $\mathfrak{P}^{\boldsymbol{t}}_{lb} = \big\{ \mathbf{s} \in \mathfrak{P}_{lb} \mid \bigwedge_{i=1}^k \mathcal{W}_i[\mathbf{s}/\overline{\alpha}] \in [a_i, b_i] \big\}$. Then $\mathfrak{P}^{\boldsymbol{t}}_{lb}$ is again a polytope and we write $\mathrm{vol}(\mathfrak{P}^{\boldsymbol{t}}_{lb})$ for its volume. The definition of $\mathfrak{P}^{\boldsymbol{t}}_{ub}$ and $\mathrm{vol}(\mathfrak{P}^{\boldsymbol{t}}_{ub})$ is analogous. As for interval traces, we call two boxes $\boldsymbol{t}_1, \boldsymbol{t}_2$ *compatible* if the intervals are almost disjoint in at least one position. A set of boxes B is *compatible* if its elements are pairwise compatible and *exhaustive* if $\bigcup_{\boldsymbol{t} \in B} \mathfrak{P}^{\boldsymbol{t}}_{lb} = \mathfrak{P}_{lb}$ and $\bigcup_{\boldsymbol{t} \in B} \mathfrak{P}^{\boldsymbol{t}}_{ub} = \mathfrak{P}_{ub}$ (cf. Section 3.3).

Proposition 6.4. *Let B be a compatible and exhaustive set of boxes. Then $\sum_{\boldsymbol{t} \in B} lb(\boldsymbol{t}) \cdot \mathrm{vol}\left(\mathfrak{P}^{\boldsymbol{t}}_{lb}\right) \leq [\![\Psi]\!]_{lb}(I)$ and $[\![\Psi]\!]_{ub}(I) \leq \sum_{\boldsymbol{t} \in B} ub(\boldsymbol{t}) \cdot \mathrm{vol}\left(\mathfrak{P}^{\boldsymbol{t}}_{ub}\right)$.*

As in the standard interval semantics, a finer partition into boxes yields more precise bounds. While the volume computation involved in Proposition 6.4 is expensive [22], the number of splits on the linear functions is much smaller than that needed in the standard interval-based semantics. Our experiments empirically demonstrate that the direct splitting of linear functions (if applicable) is usually superior to the standard splitting. In GuBPI we compute a set of exhaustive boxes by first computing a lower and upper bound on each $\mathcal{W}_i \in \Xi$ over $\mathfrak{P}_{lb}$ (or $\mathfrak{P}_{ub}$) by solving a linear program (LP) and splitting the resulting range in evenly sized chunks.

Beyond uniform samples and linear scores. We can extend our linear optimization to non-uniform samples and arbitrary symbolic values in Ξ. We accomplish the former by *combining* the optimised semantics (where we bound linear expressions) with the standard interval-trace semantics (where we bound individual sample variables). For the latter, we can identify linear sub-expressions of the expressions in Ξ, use boxes to bound each linear sub-expression, and use interval arithmetic to infer bounds on the entire expression from bounds on its linear sub-expressions. More details can be found in the full version [5].

Example 6.3. Consider the path $\Psi = (3\alpha_1, 5, \Delta, \Xi)$ derived in Example 6.2. We use 1-dimensional boxes to bound $\alpha_2 + \alpha_4$ (the single linear sub-expression of the symbolic values in Ξ). To obtain a lower bound on $[\![\Psi]\!]_{lb}(I)$, we sum over all boxes $\boldsymbol{t} = \langle [a_1, b_1] \rangle$ and take the product of $\mathrm{vol}\left(\mathfrak{P}^{\boldsymbol{t}}_{lb}\right)$ with the lower interval bound of $\mathrm{pdf}_{\mathrm{Normal}(1.1,0.1)}([a_1, b_1] + [0, \infty])$ (evaluated in interval arithmetic). Analogously, for the upper bound we take the product of $\mathrm{vol}\left(\mathfrak{P}^{\boldsymbol{t}}_{ub}\right)$ with the upper interval bound of $\mathrm{pdf}_{\mathrm{Normal}(1.1,0.1)}([a_1, b_1] + [0, \infty])$.

7 Practical Evaluation

We have implemented our semantics in the prototype GuBPI (gubpi-tool.github.io), written in F#. In cases where we apply the linear optimisation of our semantics, we use Vinci [9] to discharge volume computations of convex polytopes. We set out to answer the following questions:

1. How does GuBPI perform on instances that could already be solved (e.g. by PSI [25])?
2. Is GuBPI able to infer useful bounds on recursive programs that could not be handled rigorously before?

7.1 Probability Estimation

We collected a suite of 18 benchmarks from [54]. Each benchmark consists of a program P and a query ϕ over the variables

[10] For example, if C denotes the function $\overline{\alpha} \mapsto \mathbf{w}^\intercal \overline{\alpha} + [a, b]$ we can transform a constraint $\forall t \in \ulcorner C[\mathbf{s}/\overline{\alpha}] \urcorner . t \leq r$ into the linear constraint $\mathbf{w}^\intercal \overline{\alpha} + b \leq r$.

[11] Note the similarity to the interval trace semantics. While the ith position in an interval trace bounds the value of the ith sample variable, the ith entry of a box bounds the ith score value.

Table 1. Evaluation on selected benchmarks from [54]. We give the times (in seconds) and bounds computed by [54] and GuBPI. Details on the exact queries (the Q column) can be found in the full version [5].

		Tool from [54]		GuBPI	
Program	**Q**	t	**Result**	t	**Result**
tug-of-war	Q1	1.29	[0.6126, 0.6227]	0.72	[0.6134, 0.6135]
tug-of-war	Q2	1.09	[0.5973, 0.6266]	0.79	[0.6134, 0.6135]
beauquier-3	Q1	1.15	[0.5000, 0.5261]	22.5	[0.4999, 0.5001]
ex-book-s	Q1	8.48	[0.6633, 0.7234]	6.52	[0.7417, 0.7418]
ex-book-s	Q2⋆	10.3	[0.3365, 0.3848]	8.01	[0.4137, 0.4138]
ex-cart	Q1	2.41	[0.8980, 1.1573]	67.3	[0.9999, 1.0001]
ex-cart	Q2	2.40	[0.8897, 1.1573]	68.5	[0.9999, 1.0001]
ex-cart	Q3	0.15	[0.0000, 0.1150]	67.4	[0.0000, 0.0001]
ex-ckd-epi-s	Q1⋆	0.15	[0.5515, 0.5632]	0.86	[0.0003, 0.0004]
ex-ckd-epi-s	Q2⋆	0.08	[0.3019, 0.3149]	0.84	[0.0003, 0.0004]
ex-fig6	Q1	1.31	[0.1619, 0.7956]	21.2	[0.1899, 0.1903]
ex-fig6	Q2	1.80	[0.2916, 1.0571]	21.4	[0.3705, 0.3720]
ex-fig6	Q3	1.51	[0.4314, 2.0155]	24.7	[0.7438, 0.7668]
ex-fig6	Q4	3.96	[0.4400, 3.0956]	27.4	[0.8682, 0.9666]
ex-fig7	Q1	0.04	[0.9921, 1.0000]	0.18	[0.9980, 0.9981]
example4	Q1	0.02	[0.1910, 0.1966]	0.31	[0.1918, 0.1919]
example5	Q1	0.06	[0.4478, 0.4708]	0.27	[0.4540, 0.4541]
herman-3	Q1	0.47	[0.3750, 0.4091]	124	[0.3749, 0.3751]

of P. We bound the probability of the event described by ϕ using the tool from [54] and GuBPI (Table 1). While our tool is generally slower than the one in [54], the completion times are still reasonable. Moreover, in many cases, the bounds returned by GuBPI are tighter than those of [54]. In addition, for benchmarks marked with a ⋆, the two pairs of bounds contradict each other.[12] We should also remark that GuBPI cannot handle all benchmarks proposed in [54] because the heavy use of conditionals causes our precise symbolic analysis to suffer from the well-documented path explosion problem [7, 10, 29]. Perhaps unsurprisingly, [54] can handle such examples much better, as one of their core contributions is a stochastic method to reduce the number of paths considered (see Section 8). Also note that [54] is restricted to uniform samples, linear guards and score-free programs, whereas we tackle a much more general problem.

7.2 Exact Inference

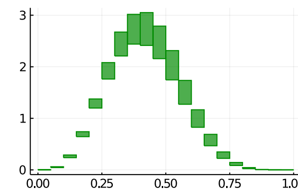
To evaluate our tool on instances that can be solved exactly, we compared it with PSI [25, 26], a symbolic solver which can, in certain cases, compute a closed-form solution of the posterior. We note that whenever exact inference is possible, exact solutions will always be superior to mere bounds and, due to the overhead of our semantics, will often be found faster. Because of the different output formats (i.e. exact results vs. bounds), a direct comparison between exact methods and GuBPI is challenging. As a consistency check, we collected benchmarks from the PSI repository where the output domain is finite and GuBPI can therefore compute *exact* results (tight bounds). They agree with PSI in all cases, which includes 8 of the 21 benchmarks from [25]. We report the computation times in Table 2.

We then considered examples where GuBPI computes non-tight bounds. For space reasons, we can only include a selection of examples in this paper. The bounds computed by GuBPI and a short description of each example are shown in Fig. 5. We can see that, despite the relatively loose bounds, they are still useful and provide the user with a rough—and most importantly, *guaranteed to be correct*—idea of the denotation.

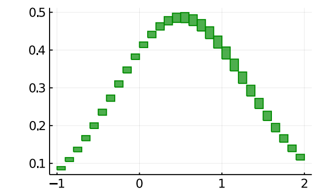
[12] A stochastic simulation using 10^6 samples in Anglican [59] yielded results that fall within GuBPI's bounds but violate those computed by [54].

Table 2. Probabilistic programs with discrete domains from PSI [25]. The times for PSI and GuBPI are given in seconds.

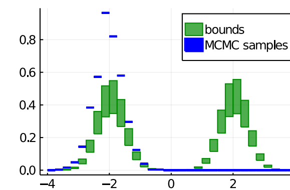
Instance	t_{PSI}	t_{GuBPI}	**Instance**	t_{PSI}	t_{GuBPI}
burglarAlarm	0.06	0.21	coins	0.04	0.18
twoCoins	0.04	0.21	ev-model1	0.04	0.21
grass	0.06	0.37	ev-model2	0.04	0.20
noisyOr	0.14	0.72	murderMystery	0.04	0.19
bertrand	0.04	0.22	coinBiasSmall	0.13	1.92
coinPattern	0.04	0.19	gossip	0.08	0.24



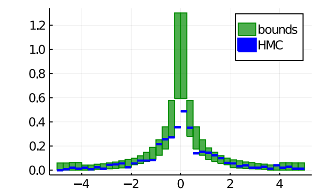
(a) `coinBias` example from [25]. The program samples a beta prior on the bias of a coin and observes repeated coin flips (26 seconds).



(b) `max` example from [25]. The program compute the maximum of two i.i.d. normal samples (31 seconds).

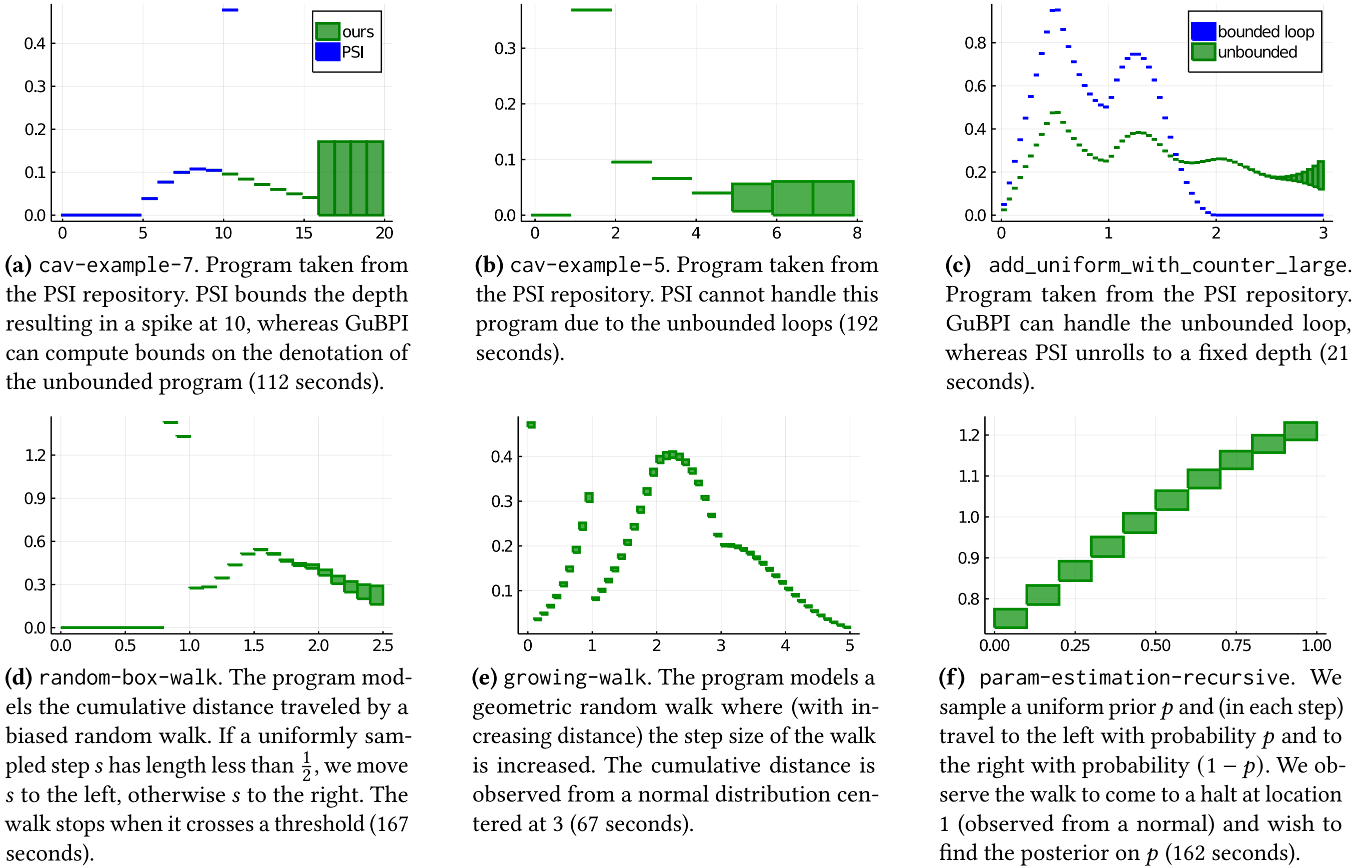


(c) Binary Gaussian Mixture Model from [63] (39 seconds). MCMC methods usually find only one mode.



(d) Neal's funnel from [33, 46] (2.8 seconds). HMC misses some probability mass around 0.

Figure 5. Guaranteed Bounds computed by GuBPI for a selection of non-recursive models from [25, 26, 46, 63].



(a) `cav-example-7`. Program taken from the PSI repository. PSI bounds the depth resulting in a spike at 10, whereas GuBPI can compute bounds on the denotation of the unbounded program (112 seconds).

(b) `cav-example-5`. Program taken from the PSI repository. PSI cannot handle this program due to the unbounded loops (192 seconds).

(c) `add_uniform_with_counter_large`. Program taken from the PSI repository. GuBPI can handle the unbounded loop, whereas PSI unrolls to a fixed depth (21 seconds).

(d) `random-box-walk`. The program models the cumulative distance traveled by a biased random walk. If a uniformly sampled step s has length less than $\frac{1}{2}$, we move s to the left, otherwise s to the right. The walk stops when it crosses a threshold (167 seconds).

(e) `growing-walk`. The program models a geometric random walk where (with increasing distance) the step size of the walk is increased. The cumulative distance is observed from a normal distribution centered at 3 (67 seconds).

(f) `param-estimation-recursive`. We sample a uniform prior p and (in each step) travel to the left with probability p and to the right with probability $(1 - p)$. We observe the walk to come to a halt at location 1 (observed from a normal) and wish to find the posterior on p (162 seconds).

Figure 6. Guaranteed bounds computed by GuBPI for a selection of recursive models.

The success of exact solvers such as PSI depends on the underlying symbolic solver (and the optimisations implemented). Consequently, there are instances where the symbolic solver cannot compute a closed-form (integral-free) solution. Conversely, while our method is (theoretically) applicable to a very broad class of programs, there exist programs where the symbolic solver finds solutions but the analysis in GuBPI becomes infeasible due to the large number of interval traces required.

7.3 Recursive Models

We also evaluated our tool on complex models that *cannot* be handled by any of the existing methods. For space reasons, we only give an overview of some examples. Unexpectedly, we found recursive models in the PSI repository: there are examples that are created by unrolling loops to a fixed depth. This fixed unrolling changes the posterior of the model. Using our method we can handle those examples *without* bounding the loop. Three such examples are shown in Figs. 6a to 6c. In Fig. 6a, PSI bounds the iterations resulting in a spike at 10 (the unrolling bound). For Fig. 6b, PSI does not provide any solution whereas GuBPI provides useful bounds. For Fig. 6c, PSI bounds the loop to compute results (displayed in blue) whereas GuBPI computes the green bounds on the unbounded program. It is obvious that the bounds differ significantly, highlighting the impact that unrolling to a fixed depth can have on the denotation. This again strengthens the claim that rigorous methods that can handle unbounded loops/recursion are needed. There also exist unbounded discrete examples where PSI computes results for the bounded version that differ from the denotation of the unbounded program. Figs. 6d to 6f depict further recursive examples (alongside a small description).

Lastly, as a *very* challenging example, we consider the pedestrian example (Example 1.1) again. The bounds computed by GuBPI are given in Fig. 7 together with the two stochastic results from Fig. 1. The bounds are clearly precise enough to rule out the HMC samples. Since this example has infinite expected running time, it is very challenging and GuBPI takes about 1.5h (84 min).[13] In fact, guaranteed bounds are the only method to recognise the wrong samples with certainty (see the next section for statistical methods).

[13]While the running time seems high, we note that Pyro HMC took about an hour to generate 10^4 samples and produce the (wrong) histogram. Diagnostic methods like simulation-based calibration took even longer (>30h) and delivered inconclusive results (see Section 7.4 for details).

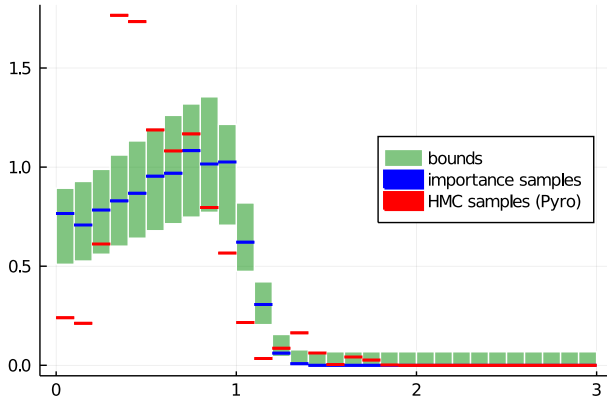


Figure 7. Bounds for the pedestrian example (Example 1.1).

Table 3. Running times of GuBPI vs SBC for (Pyro's) HMC. Times are given in seconds (s) and hours (h).

Instance	t_{GuBPI}	t_{SBC}
Binary GMM (1-dimensional) (Fig. 5c)	39s	1h
Binary GMM (2-dimensional)	4h	1.5h
Pedestrian Example (Fig. 7)	1.5h	>300h

7.4 Comparison with Statistical Validation

A general approach to validating inference algorithms for a generative Bayesian model is *simulation-based calibration* (SBC) [13, 58]. SBC draws a sample θ from the prior distribution of the parameters, generates data y for these parameters, and runs the inference algorithm to produce posterior samples $\theta_1, \ldots, \theta_L$ given y. If the posterior samples follow the true posterior distribution, the rank statistic of the prior sample θ relative to the posterior samples will be uniformly distributed. If the empirical distribution of the rank statistic after many such simulations is non-uniform, this indicates a problem with the inference. While SBC is very general, it is computationally expensive because it performs inference in every simulation. Moreover, as SBC is a stochastic validation approach, any fixed number of samples may fail to diagnose inference errors that only occur on a very low probability region.

We compare the running times of GuBPI and SBC for three examples where Pyro's HMC yields wrong results (Table 3). Running SBC on the pedestrian example (with a reduced sample size and using the parameters recommended in [58]) took 32 hours and was still inconclusive because of strong autocorrelation. Reducing the latter via thinning requires more samples, and would increase the running time to >300 hours. Similarly, GuBPI diagnoses the problem with the mixture model in Fig. 5c in significantly less time than SBC. However, for higher-dimensional versions of this mixture model, SBC clearly outperforms GuBPI. We give a more detailed discussion of SBC for these examples in the full version [5].

7.5 Limitations and Future Improvements

The theoretical foundations of our interval-based semantics ensure that GuBPI is applicable to a very broad class of programs (cf. Section 4). In practice, as usual for exact methods, GuBPI does not handle all examples equally well.

Firstly, as we already saw in Section 7.1, the symbolic execution—which forms the entry point of the analysis—suffers from path explosion. On some extreme loop/recursion-free programs (such as example-ckd-epi from [54]), our tool cannot compute all (finitely many) symbolic paths in reasonable time, let alone analyse them in our semantics. Extending the approach from [54], to sample representative program paths (in the presence of conditioning), is an interesting future direction that we can combine with the rigorous analysis provided by our interval type system.

Secondly, our interval-based semantics imposes bounds on each sampled variable and thus scales exponentially with the dimension of the model; this is amplified in the case where the optimised semantics (Section 6.4) is not applicable. It would be interesting to explore whether this can be alleviated using different trace splitting techniques.

Lastly, the bounds inferred by our interval type system take the form of a single interval with no information about the exact distribution on that interval. For example, the most precise bound derivable for the term $\mu_x^{\varphi}.x \oplus \left[\varphi(x + \text{sample}) \oplus \varphi(x - \text{sample})\right]$ is $[a,b] \to \left\{ {}^{[-\infty,\infty]}_{[1,1]} \right\}$ for any a, b. After unrolling to a fixed depth, the approximation of the paths not terminating within the fixed depth is therefore imprecise. For future work, it would be interesting to improve the bounds in our type system to provide more information about the distribution by means of rigorous approximations of the denotation of the fixpoint in question (i.e. a probabilistic summary of the fixpoint [44, 48, 61]).

8 Related Work

Interval trace semantics and Interval SPCF. Our interval trace semantics to compute bounds on the denotation is similar to the semantics introduced by Beutner and Ong [4], who study an interval approximation to obtain *lower* bounds on the termination probability. By contrast, we study the more challenging problem of bounding the program denotation which requires us to track the weight of an execution, and to prove that the denotation approximates a Lebesgue integral, which requires novel proof ideas. Moreover, whereas the termination probability of a program is always upper bounded by 1, here we derive both lower and *upper* bounds.

Probability estimation. Sankaranarayanan et al. [54] introduced a static analysis framework to infer bounds on a class of definable events in (*score-free*) probabilistic programs. The idea of their approach is that if we find a finite set $\mathcal{T}$ of symbolic traces with cumulative probability at least $1 - c$, and a given event ϕ occurs with probability at most b on the

traces in $\mathcal{T}$, then ϕ occurs with probability at most $b+c$ on the entire program. In the presence of conditioning, the problem becomes vastly more difficult, as the aggregate weight on the unexplored paths can be unbounded, giving ∞ as the only derivable upper bound (and therefore also ∞ as the best upper bound on the normalising constant). In order to infer guaranteed bounds, it is necessary to analyse *all* paths in the program, which we accomplish via static analysis and in particular our interval type system. The approach from [54] was extended by Albarghouthi et al. [1] to compute the probability of events defined by arbitrary SMT constraints but is restricted to score-free and non-recursive programs. Our interval-based approach, which may be seen as a variant of theirs, is founded on a complete semantics (Theorem 4.3), can handle recursive program with (soft) scoring, and is applicable to a broad class of primitive functions.

Note that we consider programs with *soft* conditioning in which scoring cannot be reduced to volume computation directly.[14] Intuitively, soft conditioning performs a (global) re-weighting of the set of traces, which cannot be captured by (local) volume computations. In our interval trace semantics, we instead track an approximation of the weight along each interval trace.

Exact inference. There are numerous approaches to inferring the exact denotation of a probabilistic program. Holtzen et al. [37] introduced an inference method to efficiently compute the denotation of programs with discrete distributions. By exploiting program structure to factorise inference, their system Dice can perform exact inference on programs with hundreds of thousands of random variables. Gehr et al. [25] introduced PSI, an exact inference system that uses symbolic manipulation and integration. A later extension, λPSI [26], adds support for higher-order functions and nested inference. The PPL Hakaru [45] supports a variety of inference algorithms on programs with both discrete and continuous distributions. Using program transformation and partial evaluation, Hakaru can perform exact inference via symbolic disintegration [55] on a limited class of programs. Saad et al. [53] introduced SPPL, a system that can compute exact answers to a range of probabilistic inference queries, by translating a restricted class of programs to sum-product expressions, which are highly effective representations for inference.

While exact results are obviously desirable, this kind of inference only works for a restricted family of programs: none of the above exact inference systems allow (unbounded) recursion. Unlike our tool, they are therefore unable to handle, for instance, the challenging Example 1.1 or the programs in Fig. 6.

[14]For programs including only hard-conditioning (i.e. scoring is only possible with 0 or 1), the posterior probability of an event φ can be computed by dividing the probability of all traces with weight 1 on which φ holds by the probability of all traces with weight 1.

Abstract interpretation. There are various approaches to probabilistic abstract interpretation, so we can only discuss a selection. Monniaux [42, 43] developed an abstract domain for (score-free) probabilistic programs given by a weighted sum of abstract regions. Smith [56] considered truncated normal distributions as an abstract domain and developed analyses restricted to score-free programs with only linear expressions. Extending both approaches to support soft conditioning is non-trivial as it requires the computation of integrals on the abstract regions. In our interval-based semantics, we abstract the concrete traces (by means of interval traces) and not the denotation. This allows us to derive bounds on the weight along the abstracted paths.

Huang et al. [38] discretise the domain of continuous samples into interval cubes and derive posterior approximations on each cube. The resulting approximation converges to the true posterior (similarly to approximate/stochastic methods) but does not provide exact/guaranteed bounds and is not applicable to recursive programs.

Refinement types. Our interval type system (Section 5) may be viewed as a type system that refines not just the value of an expression but also its weight [23]. To our knowledge, no existing type refinement system can bound the weight of program executions. Moreover, the seamless integration with our interval trace semantics by design allows for much cheaper type inference, without resorting to an SMT or Horn constraint solver. This is a crucial advantage since a typical GuBPI execution queries the analysis numerous times.

Stochastic methods. A general approach to validating inference algorithms for a generative Bayesian model is *simulation-based calibration* (SBC) [13, 58], discussed in Section 7.4. Grosse et al. [35] introduced a method to estimate the log marginal likelihood of a model by constructing stochastic lower/upper bounds. They show that the true value can be sandwiched between these two stochastic bounds with high probability. In closely related work [17, 34], this was applied to measure the accuracy of approximate probabilistic inference algorithms on a specified dataset. By contrast, our bounds are non-stochastic and our method is applicable to arbitrary programs of a universal PPL.

9 Conclusion

We have studied the problem of inferring guaranteed bounds on the posterior of programs written in a universal PPL. Our work is based on the interval trace semantics, and our weight-aware interval type system gives rise to a tool that can infer useful bounds on the posterior of interesting recursive programs. This is a capability beyond the reach of existing methods, such as exact inference. As a method of Bayesian inference for statistical probabilistic programs, we can view our framework as occupying a useful middle ground between approximate stochastic inference and exact inference.

References

[1] Aws Albarghouthi, Loris D'Antoni, Samuel Drews, and Aditya V. Nori. 2017. FairSquare: probabilistic verification of program fairness. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017). https://doi.org/10.1145/3133904

[2] Velleda Baldoni, Nicole Berline, Jesus De Loera, Matthias Köppe, and Michèle Vergne. 2011. How to integrate a polynomial over a simplex. *Math. Comp.* 80, 273 (2011). https://doi.org/10.1090/S0025-5718-2010-02378-6

[3] Atilim Günes Baydin, Lei Shao, Wahid Bhimji, Lukas Alexander Heinrich, Lawrence Meadows, Jialin Liu, Andreas Munk, Saeid Naderiparizi, Bradley Gram-Hansen, Gilles Louppe, Mingfei Ma, Xiaohui Zhao, Philip H. S. Torr, Victor W. Lee, Kyle Cranmer, Prabhat, and Frank Wood. 2019. Etalumis: bringing probabilistic programming to scientific simulators at scale. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2019*. ACM. https://doi.org/10.1145/3295500.3356180

[4] Raven Beutner and Luke Ong. 2021. On Probabilistic Termination of Functional Programs with Continuous Distributions. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2021*. ACM. https://doi.org/10.1145/3453483.3454111

[5] Raven Beutner, Luke Ong, and Fabian Zaiser. 2022. Guaranteed Bounds for Posterior Inference in Universal Probabilistic Programmings. *CoRR* abs/2204.02948 (2022). https://doi.org/10.48550/arXiv.2204.02948

[6] 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. *Journal of Machine Learning Research* 20 (2019). http://jmlr.org/papers/v20/18-403.html

[7] Peter Boonstoppel, Cristian Cadar, and Dawson R. Engler. 2008. RWset: Attacking Path Explosion in Constraint-Based Test Generation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2008*. Springer. https://doi.org/10.1007/978-3-540-78800-3_27

[8] Johannes Borgström, Ugo Dal Lago, Andrew D. Gordon, and Marcin Szymczak. 2016. A lambda-calculus foundation for universal probabilistic programming. In *ACM SIGPLAN International Conference on Functional Programming, ICFP 2016*. ACM. https://doi.org/10.1145/2951913.2951942

[9] Benno Büeler, Andreas Enge, and Komei Fukuda. 2000. Exact Volume Computation for Polytopes: A Practical Study. In *Polytopes—combinatorics and computation*. DMV Sem., Vol. 29. Birkhäuser, Basel. https://doi.org/10.1007/978-3-0348-8438-9_6

[10] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. 2008. EXE: Automatically Generating Inputs of Death. *ACM Transactions on Information and System Security* 12, 2 (2008). https://doi.org/10.1145/1455518.1455522

[11] Arun Tejasvi Chaganty, Aditya V. Nori, and Sriram K. Rajamani. 2013. Efficiently Sampling Probabilistic Programs via Program Analysis. In *International Conference on Artificial Intelligence and Statistics, AISTATS 2013 (JMLR, Vol. 31)*. http://proceedings.mlr.press/v31/chaganty13a.html

[12] Adrien Champion, Tomoya Chiba, Naoki Kobayashi, and Ryosuke Sato. 2020. ICE-Based Refinement Type Discovery for Higher-Order Functional Programs. *Journal of Automated Reasoning* 64, 7 (2020). https://doi.org/10.1007/s10817-020-09571-y

[13] Samantha R Cook, Andrew Gelman, and Donald B Rubin. 2006. Validation of software for Bayesian models using posterior quantiles. *Journal of Computational and Graphical Statistics* 15, 3 (2006). https://doi.org/10.1198/106186006X136976

[14] Patrick Cousot and Radhia Cousot. 1976. Static determination of dynamic properties of programs. In *International Symposium on Programming*. Dunod.

[15] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *ACM Symposium on Principles of Programming Languages, POPL 1977*. ACM. https://doi.org/10.1145/512950.512973

[16] Ryan Culpepper and Andrew Cobb. 2017. Contextual Equivalence for Probabilistic Programs with Continuous Random Variables and Scoring. In *European Symposium on Programming, ESOP 2017 (LNCS, Vol. 10201)*. Springer. https://doi.org/10.1007/978-3-662-54434-1_14

[17] Marco F. Cusumano-Towner and Vikash K. Mansinghka. 2017. AIDE: An algorithm for measuring the accuracy of probabilistic inference algorithms. In *Annual Conference on Neural Information Processing Systems, NIPS 2017*. https://proceedings.neurips.cc/paper/2017/hash/acab0116c354964a558e65bdd07ff047-Abstract.html

[18] 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 International Conference on Programming Language Design and Implementation, PLDI 2019*. ACM. https://doi.org/10.1145/3314221.3314642

[19] Hend Dawood. 2011. *Theories of Interval Arithmetic: Mathematical Foundations and Applications.* LAP Lambert Academic Publishing.

[20] Jesús A De Loera, Brandon Dutra, Matthias Koeppe, Stanislav Moreinis, Gregory Pinto, and Jianqiu Wu. 2013. Software for exact integration of polynomials over polyhedra. *Computational Geometry* 46, 3 (2013). https://doi.org/10.1016/j.comgeo.2012.09.001

[21] Simon Duane, Anthony D Kennedy, Brian J Pendleton, and Duncan Roweth. 1987. Hybrid Monte Carlo. *Physics letters B* 195, 2 (1987). https://doi.org/10.1016/0370-2693(87)91197-X

[22] Martin E. Dyer and Alan M. Frieze. 1988. On the Complexity of Computing the Volume of a Polyhedron. *SIAM J. Comput.* 17, 5 (1988). https://doi.org/10.1137/0217060

[23] Timothy S. Freeman and Frank Pfenning. 1991. Refinement Types for ML. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 1991*. ACM. https://doi.org/10.1145/113445.113468

[24] Hong Ge, Kai Xu, and Zoubin Ghahramani. 2018. Turing: A Language for Flexible Probabilistic Inference. In *International Conference on Artificial Intelligence and Statistics, AISTATS 2018 (PMLR, Vol. 84)*. https://proceedings.mlr.press/v84/ge18b.html

[25] Timon Gehr, Sasa Misailovic, and Martin T. Vechev. 2016. PSI: Exact Symbolic Inference for Probabilistic Programs. In *International Conference on Computer Aided Verification, CAV 2016 (LNCS, Vol. 9779)*. Springer. https://doi.org/10.1007/978-3-319-41528-4_4

[26] Timon Gehr, Samuel Steffen, and Martin T. Vechev. 2020. λPSI: exact inference for higher-order probabilistic programs. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020*. ACM. https://doi.org/10.1145/3385412.3386006

[27] Jaco Geldenhuys, Matthew B. Dwyer, and Willem Visser. 2012. Probabilistic symbolic execution. In *International Symposium on Software Testing and Analysis, ISSTA 2012*. ACM. https://doi.org/10.1145/2338965.2336773

[28] Zoubin Ghahramani. 2013. Bayesian non-parametrics and the probabilistic approach to modelling. *Philosophical Transactions of the Royal Society A* 371 (2013). https://doi.org/10.1098/rsta.2011.0553

[29] Patrice Godefroid. 2007. Compositional dynamic test generation. In *ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2007*. ACM. https://doi.org/10.1145/1190216.1190226

[30] Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. 2008. Church: a language for generative models. In *Conference on Uncertainty in Artificial Intelligence, UAI 2008*. AUAI Press.

[31] Noah D. Goodman and Andreas Stuhlmüller. 2014. *The Design and Implementation of Probabilistic Programming Languages.*

[32] Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. 2014. Probabilistic programming. In *Future of Software Engineering, FOSE 2014*. ACM. https://doi.org/10.1145/2593882.2593900

[33] Maria I. Gorinova, Dave Moore, and Matthew D. Hoffman. 2020. Automatic Reparameterisation of Probabilistic Programs. In *International Conference on Machine Learning, ICML 2020 (PMLR, Vol. 119)*. http://proceedings.mlr.press/v119/gorinova20a.html

[34] Roger B. Grosse, Siddharth Ancha, and Daniel M. Roy. 2016. Measuring the reliability of MCMC inference with bidirectional Monte Carlo. In *Annual Conference on Neural Information Processing Systems, NIPS 2016*. https://proceedings.neurips.cc/paper/2016/hash/0e9fa1f3e9e66792401a6972d477dcc3-Abstract.html

[35] Roger B. Grosse, Zoubin Ghahramani, and Ryan P. Adams. 2015. Sandwiching the marginal likelihood using bidirectional Monte Carlo. *CoRR* abs/1511.02543 (2015). https://doi.org/10.48550/arXiv.1511.02543

[36] N. L. Hjort, Chris Homes, Peter Muller, and Stephen G. Walker. 2010. *Bayesian Nonparametrics*. Cambridge University Press. https://doi.org/10.1017/CBO9780511802478

[37] Steven Holtzen, Guy Van den Broeck, and Todd D. Millstein. 2020. Scaling exact inference for discrete probabilistic programs. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020). https://doi.org/10.1145/3428208

[38] Zixin Huang, Saikat Dutta, and Sasa Misailovic. 2021. AQUA: Automated Quantized Inference for Probabilistic Programs. In *International Symposium on Automated Technology for Verification and Analysis, ATVA 2021 (LNCS, Vol. 12971)*. Springer. https://doi.org/10.1007/978-3-030-88885-5_16

[39] Carol Mak, C.-H. Luke Ong, Hugo Paquet, and Dominik Wagner. 2021. Densities of Almost Surely Terminating Probabilistic Programs are Differentiable Almost Everywhere. In *European Symposium on Programming, ESOP 2021 (LNCS, Vol. 12648)*. Springer. https://doi.org/10.1007/978-3-030-72019-3_16

[40] Carol Mak, Fabian Zaiser, and Luke Ong. 2021. Nonparametric Hamiltonian Monte Carlo. In *International Conference on Machine Learning, ICML 2021 (PMLR, Vol. 139)*. http://proceedings.mlr.press/v139/mak21a.html

[41] Chris Manning and Hinrich Schütze. 1999. *Foundations of Statistical Natural Language Processing*. MIT Press. Cambridge, MA.

[42] David Monniaux. 2000. Abstract Interpretation of Probabilistic Semantics. In *International Symposium on Static Analysis, SAS 2000 (LNCS, Vol. 1824)*. Springer. https://doi.org/10.1007/978-3-540-45099-3_17

[43] David Monniaux. 2001. An abstract Monte-Carlo method for the analysis of probabilistic programs. In *ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2001*. ACM. https://doi.org/10.1145/360204.360211

[44] Markus Müller-Olm and Helmut Seidl. 2004. Precise interprocedural analysis through linear algebra. In *ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2004*. ACM. https://doi.org/10.1145/964001.964029

[45] Praveen Narayanan, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov. 2016. Probabilistic Inference by Program Transformation in Hakaru (System Description). In *International Symposium on Functional and Logic Programming, FLOPS 2016 (LNCS, Vol. 9613)*. Springer. https://doi.org/10.1007/978-3-319-29604-3_5

[46] Radford M Neal. 2003. Slice sampling. *The annals of statistics* 31, 3 (2003). https://doi.org/10.1214/aos/1056562461

[47] Art B. Owen. 2013. *Monte Carlo theory, methods and examples*.

[48] Andreas Podelski, Ina Schaefer, and Silke Wagner. 2005. Summaries for While Programs with Recursion. In *European Symposium on Programming, ESOP 2005 (LNCS, Vol. 3444)*. Springer. https://doi.org/10.1007/978-3-540-31987-0_8

[49] David Pollard. 2002. *A User's Guide to Measure-Theoretic Probability*. Cambridge University Press. https://doi.org/10.1017/CBO9780511811555

[50] Fredrik Ronquist, Jan Kudlicka, Viktor Senderov, Johannes Borgström, Nicolas Lartillot, Daniel Lundén, Lawrence Murray, Thomas B Schön, and David Broman. 2021. Universal probabilistic programming offers a powerful approach to statistical phylogenetics. *Communications biology* 4, 1 (2021). https://doi.org/10.1038/s42003-021-01753-7

[51] Vivekananda Roy. 2020. Convergence Diagnostics for Markov Chain Monte Carlo. *Annual Review of Statistics and Its Application* 7 (2020). https://doi.org/10.1146/annurev-statistics-031219-041300

[52] Reuven Y. Rubinstein and Dirk P. Kroese. 2017. *Simulation and the Monte Carlo Method* (3rd ed.). Wiley. https://doi.org/10.1002/9781118631980

[53] Feras A. Saad, Martin C. Rinard, and Vikash K. Mansinghka. 2021. SPPL: probabilistic programming with fast exact symbolic inference. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2021*. ACM. https://doi.org/10.1145/3453483.3454078

[54] Sriram Sankaranarayanan, Aleksandar Chakarov, and Sumit Gulwani. 2013. Static analysis for probabilistic programs: inferring whole program properties from finitely many paths. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2013*. ACM. https://doi.org/10.1145/2491956.2462179

[55] Chung-chieh Shan and Norman Ramsey. 2017. Exact Bayesian inference by symbolic disintegration. In *ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017*. ACM. https://doi.org/10.1145/3009837.3009852

[56] Michael J. A. Smith. 2008. Probabilistic Abstract Interpretation of Imperative Programs using Truncated Normal Distributions. *Electronic Notes in Theoretical Computer Science* 220, 3 (2008). https://doi.org/10.1016/j.entcs.2008.11.018

[57] Sam Staton. 2017. Commutative Semantics for Probabilistic Programming. In *European Symposium on Programming, ESOP 2017 (LNCS, Vol. 10201)*. Springer. https://doi.org/10.1007/978-3-662-54434-1_32

[58] Sean Talts, Michael Betancourta, Daniel Simpson, Aki Vehtari, and Andrew Gelman. 2018. Validating Bayesian Inference Algorithms with Simulation-Based Calibration. *arXiv* 1804.06788 (2018). https://doi.org/10.48550/arXiv.1804.06788

[59] David Tolpin, Jan-Willem van de Meent, and Frank D. Wood. 2015. Probabilistic Programming in Anglican. In *European Conference on Machine Learning and Knowledge Discovery in Databases, ECML PKDD 2015 (LNCS, Vol. 9286)*. Springer. https://doi.org/10.1007/978-3-319-23461-8_36

[60] Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. 2018. An Introduction to Probabilistic Programming. *CoRR* abs/1809.10756 (2018). https://doi.org/10.48550/arXiv.1809.10756

[61] Di Wang, Jan Hoffmann, and Thomas W. Reps. 2018. PMAF: an algebraic framework for static analysis of probabilistic programs. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2018*. ACM. https://doi.org/10.1145/3192366.3192408

[62] Cheng Zhang, Judith Bütepage, Hedvig Kjellström, and Stephan Mandt. 2019. Advances in Variational Inference. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 41, 8 (2019). https://doi.org/10.1109/TPAMI.2018.2889774

[63] Yuan Zhou, Bradley J. Gram-Hansen, Tobias Kohn, Tom Rainforth, Hongseok Yang, and Frank Wood. 2019. LF-PPL: A Low-Level First Order Probabilistic Programming Language for Non-Differentiable Models. In *International Conference on Artificial Intelligence and Statistics, AISTATS 2019 (PMLR, Vol. 89)*. http://proceedings.mlr.press/v89/zhou19b.html