PSI: Exact Symbolic Inference for Probabilistic Programs

Timon Gehr[1], Sasa Misailovic[1,2], and Martin Vechev[1]

[1] ETH Zurich
[2] University of Illinois at Urbana-Champaign

Abstract. Probabilistic inference is a key mechanism for reasoning about probabilistic programs. Since exact inference is theoretically expensive, most probabilistic inference systems today have adopted approximate inference techniques, which trade precision for better performance (but often without guarantees). As a result, while desirable for its ultimate precision, the practical effectiveness of exact inference for probabilistic programs is mostly unknown.

This paper presents PSI (http://www.psisolver.org), a novel symbolic analysis system for exact inference in probabilistic programs with both continuous and discrete random variables. PSI computes succinct symbolic representations of the joint posterior distribution represented by a given probabilistic program. PSI can compute answers to various posterior distribution, expectation and assertion queries using its own backend for symbolic reasoning.

Our evaluation shows that PSI is more effective than existing exact inference approaches: (i) it successfully computed a precise result for more programs, and (ii) simplified expressions that existing computer algebra systems (e.g., Mathematica, Maple) fail to handle.

1 Introduction

Many statistical learning applications make decisions under uncertainty. Probabilistic languages provide a natural way to model uncertainty by representing complex probability distributions as programs [20,21,19,8,12,25,39,46,32]. Exact probabilistic inference for programs with only discrete random variables is already a #P-hard computational problem [13]. Programs which have both discrete and continuous variables reveal additional challenges, such as representing discrete and continuous components of the joint distribution, computing integrals, and managing a large number of terms in the joint distribution.

For these reasons, most existing probabilistic languages implement inference algorithms that calculate numerical approximations. The general approaches include sampling-based Monte Carlo methods [20,40,21,19,39,46,32] and projections to convenient probability distributions, such as variational inference [34,8] or discretization [31,12]. While these methods scale well, they typically come with no accuracy guarantees, since providing such guarantees is NP-hard [15].

At the same time, there has been a renewed research interest in symbolic inference methods due to their promise for computing more precise inference results. Existing symbolic inference works fall into different categories:

- *Approximate symbolic inference:* Several analyses of graphical models approximate continuous distribution functions with a mixture of base functions, such as truncated exponentials or polynomials, which can be integrated more easily [44,11,36,45,43]. For instance, SVE [43] approximates distributions as piecewise low-rank polynomials.
- *Interactive symbolic inference:* A user can write down the inference steps within modern computer algebra systems, such as Mathematica [3] and Maple [1]. These tools can help the user by automating parts of the integration and/or simplification of distribution expressions.
- *Exact symbolic inference:* Bhat et al. [7] presents a type-based analysis translating programs with mixed discrete/continuous variables into symbolic distribution expressions, but does not simplify integral terms symbolically and instead computes them using a numerical integration library. Most recently, Hakaru [38,10] optimizes probabilistic programs by translating a program into a distribution expression in a DSL within Maple's expression language, and simplifying this expression utilizing Maple's engine, before running (if necessary) the optimized program within a MCMC simulation.

While these works are promising steps, the practical effectiveness of exact symbolic inference in hybrid probabilistic models (with *both* discrete and continuous distributions) remains unknown, dictating the need for further investigation.

This work. We present the PSI (Probabilistic Symbolic Inference) system, a comprehensive approach for automating exact probabilistic inference via program analysis. PSI's analysis performs an *end-to-end symbolic inference* for probabilistic programs with discrete and/or continuous random variables. PSI analyzes a probabilistic program using a symbolic domain which captures the program's probability distribution in a precise manner. PSI comes with its own symbolic optimization engine which generates compact expressions that represent *joint probability density functions* using various optimizations, including algebraic simplification and symbolic integration. The symbolic domain and optimizations are designed to strike a balance between the expressiveness of the probability density expressions and the efficiency of automatically computing integrals and generating compact densities.

Our symbolic analysis (Section 3) generalizes existing analyzers for exact inference on discrete programs (e.g., those that operate at the level of concrete states [12]). Our optimization engine (Section 4) can also automatically simplify many integrals in density expressions and thus directly improve the performance of works that generate unoptimized expressions, such as [7], without requiring the full complexity of a general computer algebra system, as in [38,10]. As a result, PSI is able to compute precise and compact inference results even when the existing approaches fail (Section 5).

Contributions. Our main contributions are:

- *Symbolic inference for programs with continuous/discrete variables:* A novel approach for fully symbolic probabilistic inference. The algorithm represents the posterior distribution within a symbolic domain.
- *Probabilistic inference system:* PSI, an implementation of our algorithm together with optimizations that simplify the symbolic representation of the posterior distribution. PSI is available at **http://www.psisolver.org**.
- *Evaluation:* The paper shows an experimental evaluation of PSI against state-of-the-art symbolic inference techniques – Hakaru, Maple, Mathematica, and SVE – on a corpus of 21 benchmarks selected from the literature. PSI produced correct and compact distribution expressions for more benchmarks than the alternatives. In addition, we compare PSI to state-of-the-art approximate inference engines, Infer.NET [34] and R2 [39], and show the benefits of exact inference.

Based on our results, we believe that PSI is the most effective exact symbolic inference engine to date and is useful for understanding the potential of exact symbolic inference for probabilistic programs.

2 Overview

Figure 1 presents the ClickGraph probabilistic program, adapted from a Fun language program from [34]. It describes an information retrieval model that calculates the posterior distribution of the similarity of two files, conditioned on the users' access patterns to these files.

The program first specifies the prior distribution on the document similarity (line 2) and the recorded accesses to `A` and `B` for each user (lines 4-5). It then specifies a trial in which the variable `sim` is the similarity of the documents for an issued query (line 7). If the documents are similar, the probabilities of accessing them (`p1` and `p2`) are the same, otherwise `p1` and `p2` are independent (lines 9-14). Finally, the variables `clickA` and `clickB` represent outcomes of the users accessing the documents (lines 16-17), and each trial produces a specific observation using the `observe` statements (lines 18-19). The return statement specifies that PSI should compute the posterior distribution of `simAll`.

2.1 Analysis

To compute the posterior distribution, PSI analyzes the probabilistic program via a symbolic domain that captures probability distributions, and applies optimizations to simplify the resulting expression after each analysis step.

Symbolic Analysis: For each statement, the analysis computes a symbolic expression that captures the program's probability distribution at that point. The analysis operates forward, starting form the beginning of the function. As a pre-processing step, the analysis unrolls all loops and lowers the array elements into a sequence of scalars or inlined constants. The state of the analysis at each

```
1   def ClickGraph(){
2     simAll := Uniform(0,1);
3
4     clicksA := [1, 1, 1, 0, 0];
5     clicksB := [1, 1, 1, 0, 0];
6     for i in [0..5) {
7         sim := Bernoulli(simAll);
8
9         p1:=0; p2:=0;
10        if sim {
11            p1 = Uniform(0,1); p2 = p1;
12        } else {
13            p1 = Uniform(0,1); p2 = Uniform(0,1);
14        }
15
16        clickA := Bernoulli(p1);
17        clickB := Bernoulli(p2);
18        observe(clickA==clicksA[i]);
19        observe(clickB==clicksB[i]);
20    }
21    return simAll;
22  }
```

Fig. 1. ClickGraph Example

program point captures (1) the correct execution of the program as a map that relates live program variables $x_1, \dots, x_n$ to a symbolic expression e representing a *probability density* of the computation at this point in the program, and (2) erroneous executions (e.g., due to an assertion violation) represented by an aggregate error probability expression $\bar{e}$.

The analysis of the first statement (line 2) identifies that the state consists of the variable `simAll`, which has the $\texttt{Uniform}(0, 1)$ distribution. In general, for $x := \texttt{Uniform}(a, b)$, the analysis generates the expression $[a \leq x] \cdot [x \leq b]/(b - a)$, which denotes the density of this distribution. The factors $[a \leq x]$ and $[x \leq b]$ are *Iverson brackets*, guard functions that equal 1 if their conditions are true, or equal 0 otherwise. Therefore, this density has a non-zero value only if $x \in [a, b]$. In particular, for `simAll`, the analysis generates $e_{\texttt{L.2}} = [0 \leq \texttt{simAll}] \cdot [\texttt{simAll} \leq 1]$.

Since the constant arrays are inlined, the analysis next processes the statement on line 7 (in the first loop iteration). The analysis adds the variable `sim` to the state, and multiplies the expression $e_{\texttt{L.2}}$ with the density function for the distribution `Bernoulli(simAll)`:

$$\begin{aligned} e_{\texttt{L.7}} &= [0 \leq \texttt{simAll}] \cdot [\texttt{simAll} \leq 1] \cdot (\texttt{simAll} \cdot \delta(1 - \texttt{sim}) + (1 - \texttt{simAll}) \cdot \delta(\texttt{sim})) \\ \bar{e}_{\texttt{L.7}} &= 0 \end{aligned}$$

The expression $e_{\texttt{L.7}}$ represents a generalized joint probability density over `simAll` and `sim`. To encode discrete distributions like Bernoulli, the analysis uses *Dirac deltas*, $\delta(e)$, which specify a distribution with point masses at the zeros of e.

Optimizations: After analyzing each statement, the analysis simplifies the generated distribution expression by applying equivalence-preserving optimizations:

- basic algebraic manipulations (e.g., in the previous expression, an optimization can distribute multiplication over addition);
- removal of factors with trivial or unsatisfiable guards (e.g., in this example the analysis checks whether a product $[0 \leq \texttt{simAll}] \cdot [\texttt{simAll} \leq 1]$ is always equal to zero, and since it is not, leaves the expression unchanged);

– symbolic integration of the distribution expressions; for instance, at the end of the each loop iteration, the analysis expression $e_{\mathtt{L.19}}$ contains several loop-local variables: `sim`, `p1`, `p2`, `clickA`, and `clickB`. The analysis integrates out these local variables because they will not be referenced by the subsequent computation. It first removes the discrete variables `sim`, `clickA`, and `clickB` by exploiting the properties of Dirac deltas. For the continuous variables `p1` and `p2`, it computes the antiderivative (indefinite integral) using PSI's integration engine, finds the integration bounds, and evaluates the antiderivative on these bounds. After the analysis of the first loop iteration, this optimization reduces the size of the distribution expression from 22 to 6 summands.

Result of the Analysis: After analyzing the entire program, the analysis produces the final posterior probability density expression for the variable `simAll`:

$$e_{L.21} \;=\; [0 \le \mathtt{simAll}] \cdot [\mathtt{simAll} \le 1] \cdot \frac{6(\mathtt{simAll}+3)^5}{3367}$$

The analysis also computes that the final error probability $\bar{e}_{L.21}$ is 0. This is the exact posterior distribution for this program. We present this posterior density function graphically in Figure 8.

2.2 Applications of PSI

PSI's source language (with conditional and bounded loop statements) has the expressive power to represent arbitrary Bayesian networks, which encode many probabilistic models relevant in practice [22]. PSI's analysis is analogous to the variable elimination algorithm for inference in graphical models. We anticipate that PSI can be successfully deployed in several contexts:

Probabilistic Inference: PSI allows a developer to specify several classes of queries. For joint posterior distribution, a user may return multiple variables in the `return` statement. The special operators `FromMarginal`(`e`) and `Expectation`(`e`) return the marginal distribution and the expectation of an expression `e`, respectively. A developer can also specify assertions using the `assert(e)` statement.

Testing and Debugging: The exact inference results produced by PSI can be used as reference versions for debugging and testing approximate inference engines. It can also be used to test existing computer algebra systems – using PSI, we found errors in Maple's simplifier (see Section 5).

Sampling from Optimized Probabilistic Programs: Optimized distribution expressions generated by PSI's symbolic optimizer can be used, in principle, for computing proposal distributions in MCMC simulations, as done by [7] and [38].

Uncertainty Propagation Analysis: PSI's analysis can serve as a basis for static analyses that propagate uncertainty through computations and determine error bars for the result. This provides a powerful alternative to existing analyses that are primarily sampling-based [9,41], with at most limited support for simplifying algebraic identities that involve random variables [41].

$$
\begin{array}{ll}
n \in \mathbb{Z} & bop \in \{+,-,*,/,\hat{}\} \quad lop \in \{\&\&, ||\} \quad cop \in \{==,\neq,<,>,\leq,\geq\} \\
r \in \mathbb{R} & \text{Dist} \in \{\texttt{Bernoulli},\texttt{Gauss},\ldots\}\ \text{SOp} \in \{\texttt{Expectation},\texttt{FromMarginal},\texttt{SampleFrom}\} \\
x \in \text{Var} & p \in \text{Prog} \to \text{Func}^{+} \\
a \in \text{ArrVar} & f \in \text{Func} \to \texttt{def}\ \ Id\texttt{(}Var^{*}\texttt{)}\ \ \{\text{Stmt}\texttt{;}\ \texttt{return}\ \text{Var}^{*}\}
\end{array}
$$

$$
\begin{array}{ll}
\text{se} \in \text{Expr} \to & n \mid r \mid x \mid a\texttt{[}\text{Expr}\texttt{]} \mid \text{Expr}\ bop\ \text{Expr} \mid \text{Expr}\ cop\ \text{Expr} \mid \text{Expr}\ lop\ \text{Expr} \mid \\
& \text{Dist}\texttt{(}\text{Expr}^{+}\texttt{)} \mid \text{SOp}\texttt{(}\text{Expr}\texttt{)} \mid f\texttt{(}\text{Expr}^{*}\texttt{)}
\end{array}
$$

$$
\begin{array}{ll}
s \in \text{Stmt} \to & x := \text{Expr} \mid a := \texttt{array}(\text{Expr}) \mid x = \text{Expr} \mid a\texttt{[}\text{Expr}\texttt{]} = \text{Expr} \mid \\
& \texttt{observe}\ \text{Expr} \mid \texttt{assert}\ \text{Expr} \mid \texttt{skip} \mid \text{Stmt}\texttt{;}\ \text{Stmt} \mid \\
& \texttt{if}\ \text{Expr}\ \{\text{Stmt}\}\ \texttt{else}\ \{\text{Stmt}\} \mid \texttt{for}\ x\ \texttt{in}\ [\text{Expr}\texttt{..}\text{Expr})\ \{\text{Stmt}\}
\end{array}
$$

Fig. 2. PSI's Source Language Syntax

3 Symbolic Inference

In this section we describe our core analysis: the procedure analyzes each statement in the program and produces a corresponding expression in our symbolic domain which captures probability distributions.

3.1 Source Language

Figure 2 presents the syntax of PSI's source language. This is a simple imperative language that operates on real-valued scalar and array data. The language defines probabilistic assignments, which can assign a random value drawn from a distribution Dist, and `observe` statements, which allow constraining the values of probabilistic expressions. The language also supports the standard sequence, conditional statement, and bounded loop statement.

3.2 Symbolic Domain for Probability Distributions

Figure 3 presents the syntax of our symbolic domain. The domain can succinctly describe joint probability distributions with discrete and continuous components:

– *Basic terms* include variables, numerical constants (such as e and π), logarithms and uninterpreted functions. These terms can form sums, products, or exponents. Division is handled using the rewrite $a/b \to a \cdot b^{-1}$.
– *Dirac deltas* represent distributions that have weight in low-dimensional sets (such as single points). In our analysis, they encode variable definitions and assignments, and linear combinations of Dirac deltas specify discrete distributions.
– *Iverson brackets* represent functions that are 1 if the condition within the brackets is satisfied and 0 otherwise. In our analysis, they encode comparison operators and certain primitive probability distributions (e.g., Uniform).
– *Integrals and infinite sums* are used during the analysis to represent marginalization of variables and UniformInt distributions respectively.
– *Gaussian antiderivative* – $(\mathrm{d}/\mathrm{d}x)^{-1}[\mathrm{e}^{-x^2}](e)$ – used to denote the function $\int_{-\infty}^{e} \mathrm{d}x\, \mathrm{e}^{-x^2}$, which cannot be decomposed into simpler elementary functions.

$$
\begin{aligned}
e \in E ::= \quad & x \mid n \mid r \mid \log(e) \mid \varphi(e_1,\ldots,e_n) \mid -e \mid e_1+\ldots+e_n \mid e_1\cdot\ldots\cdot e_n \mid e_1^{e_2} \mid \\
& \delta(e) \mid [e_1 = e_2] \mid [e_1 \le e_2] \mid [e_1 \ne e_2] \mid [e_1 < e_2] \mid \\
& \sum_{x\in\mathbb{Z}} e[\![x]\!] \mid \int_{\mathbb{R}} \mathrm{d}x\, e[\![x]\!] \mid (\mathrm{d}/\mathrm{d}x)^{-1}[\mathrm{e}^{-x^2}](e)
\end{aligned}
$$

Fig. 3. Symbolic domain for probability distributions

We use the notation $e[\![x_1,\ldots,x_n]\!]$ to denote that a symbolic distribution expression e may contain free variables $x_1,\ldots,x_n$ that are bound by an outer operator (such as a sum or integral).

Our design of the symbolic domain aims to strike a balance between *expressiveness* – the kinds of distributions it can represent – and *efficiency* – the ability of the analysis to automatically integrate functions and find simple equivalent expressions. In particular, our symbolic domain enables us to define most discrete and continuous distributions from the exponential family and other well-known primitive distributions, such as Student-t and Laplace (see the Appendix A).

Primitive Distributions: For each primitive distribution Dist, we define two mappings, $\mathrm{PDF}_{\mathrm{Dist}}$, and $\mathrm{Conditions}_{\mathrm{Dist}}$ to respectively specify the probability density function, and valid parameter and input ranges. For instance, the Bernoulli distribution with a parameter e_p has $\mathrm{PDF}_{\mathrm{Bern}}(x,e_p) = e_p\cdot\delta(1-x)+(1-e_p)\cdot\delta(x)$ and $\mathrm{Conditions}_{\mathrm{Bern}} = [0 \le e_p]\cdot[e_p \le 1]$. We present the encodings of several other primitive distributions in the Appendix A. Additionally, PSI allows the developer to specify an arbitrary density function of the resulting distribution using the `SampleFrom (sym_expr,...)` primitive, which takes as inputs a distribution expression and a set of its parameters.

Program State: A symbolic program state σ denotes a probability distribution over the program variables with an additional error state:

$$
\sigma \in \Sigma ::= \lambda M.\,\textbf{case}\ M\ \textbf{of}\ (x_1,\ldots,x_n) \Rightarrow e_1[\![x_1,\ldots,x_n]\!],\ \bot \Rightarrow e_2 \qquad (1)
$$

In a regular execution, the state is represented with the variables $x_1,\ldots,x_n$ and the posterior distribution expression e_1. We represent the error state as a symbol $\bot$ and the expression for the probability of error e_2. Conceptually, the map σ associates a probability density with each concrete program state M, which is either a tuple of values of program variables or the error state.

3.3 Analysis of Expressions

Figure 4 presents the analysis of expressions. The function A_e converts each expression of the source language to a transformer $t \in \Sigma \to \Sigma \times E$ on the symbolic representation. The transformer returns both a new state ($\sigma \in \Sigma$) and a result of expression evaluation ($e \in E$), thus capturing side effects (e.g., sampling values from probability distributions or exhibiting errors such as division by zero).

$$
\begin{aligned}
A_e :\ \mathrm{Expr} \to & (\Sigma \to \Sigma \times E)\\
A_e(x) :=& \lambda\sigma.\,(\sigma, x)\\
A_e(\mathrm{se}_1\ \mathit{bop}\ \mathrm{se}_2) :=& \lambda\sigma.\,\textbf{let}\ (\sigma_1, e_1) = A_e(\mathrm{se}_1)(\sigma)\ \textbf{and}\ (\sigma_2, e_2) = A_e(\mathrm{se}_2)(\sigma_1)\\
& \quad\textbf{in}\ (\sigma_2, e_1\ \mathit{SymbolicOp(bop)}\ e_2), \qquad \mathit{bop} \in \{\texttt{+}, \texttt{-}, \texttt{*}\}\\
A_e(\mathrm{se}_1/\mathrm{se}_2) :=& \lambda\sigma.\,\textbf{let}\ (\sigma_1, e_1) = A_e(\mathrm{se}_1)(\sigma)\ \textbf{and}\ (\sigma_2, e_2) = A_e(\mathrm{se}_2)(\sigma_1)\\
& \quad\textbf{in}\ (\mathrm{Assert}([e_2 \neq 0])(\sigma_2), e_1 \cdot {e_2}^{-1})\\
A_e(\mathrm{se}_1 \&\& \mathrm{se}_2) :=& \lambda\sigma.\,\textbf{let}\ (\sigma_1, e_1) = A_e(\mathrm{se}_1)(\sigma)\ \textbf{and}\ (\sigma_2, e_2) = A_e(\mathrm{se}_2)(\sigma_1)\\
& \quad\textbf{in}\ (\sigma_2, [e_1 \neq 0] \cdot [e_2 \neq 0])\\
A_e(\mathrm{se}_1\ ||\ \mathrm{se}_2) :=& \lambda\sigma.\,\textbf{let}\ (\sigma_1, e_1) = A_e(\mathrm{se}_1)(\sigma)\ \textbf{and}\ (\sigma_2, e_2) = A_e(\mathrm{se}_2)(\sigma_1)\\
& \quad\textbf{in}\ (\sigma_2, [\,[e_1 \neq 0] + [e_2 \neq 0] \neq 0\,])\\
A_e(\mathrm{Dist}(\mathrm{se}_1, \ldots, \mathrm{se}_n)) :=& \lambda\sigma.\,\textbf{let}\ (\sigma_1, [e_1, ..., e_n]) = A_e^*([\mathrm{se}_1, ..., \mathrm{se}_n])(\sigma)\ \textbf{and}\ \mathrm{FreshVar}(\tau)\\
& \quad\textbf{and}\ (P, C) = (\mathrm{PDF}_{\mathrm{Dist}}(\tau, e_1, ..., e_n), \mathrm{Conditions}_{\mathrm{Dist}}(e_1, ..., e_n))\\
& \quad\textbf{in let}\ \sigma_2 = (\mathrm{Distribute}(\tau, P) \circ \mathrm{Assert}(C))(\sigma_1)\ \textbf{in}\ (\sigma_2, \tau),
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{Assert}(e[\![x_1, \ldots, x_n]\!])(\lambda M.\,\textbf{case}\ M\ \textbf{of}\ (x_1, \ldots, x_n) \Rightarrow e_1[\![x_1, \ldots, x_n]\!]\,, \bot \Rightarrow e_2) :=\\
&\quad \lambda M.\,\textbf{case}\ M\ \textbf{of}\ (x_1, \ldots, x_n) \Rightarrow (e_1 \cdot [e \neq 0])[\![x_1, \ldots, x_n]\!]\,,\\
&\qquad\qquad\qquad\qquad \bot \Rightarrow e_2 + \mathrm{MarginalizeAll}([e = 0] \cdot e_1)\\
&\mathrm{Distribute}(x, e[\![x_1, ..., x_n, x]\!])(\lambda M.\,\textbf{case}\ M\ \textbf{of}\ (x_1, ..., x_n) \Rightarrow e_1[\![x_1, ..., x_n]\!]\,, \bot \Rightarrow e_2) :=\\
&\quad \lambda M.\,\textbf{case}\ M\ \textbf{of}\ (x_1, ..., x_n, x) \Rightarrow e_1[\![x_1, ..., x_n]\!] \cdot e[\![x_1, ..., x_n, x]\!]\,,\ \bot \Rightarrow e_2
\end{aligned}
$$

Fig. 4. Symbolic Analysis of Expressions

Operations: The first five rules transform source language variables to distribution expression variables (including operators via the helper function *SymbolicOp*). The rules are standard, with boolean constants `true` and `false` encoded as numbers 1 and 0, respectively. The rules compose the side effects of the operands. The division rule additionally uses the Assert helper function to add the guard $[e_2 \neq 0]$ to the distribution expression and aggregate the probability of $e_2 = 0$ to the overall error probability.

Distribution Sampling: The expression $\mathrm{Dist}(\mathrm{se}_1, \ldots, \mathrm{se}_n)$ accepts distribution parameters $\mathrm{se}_1, \ldots, \mathrm{se}_n$, which can be arbitrary expressions. For a primitive distribution Dist, the analysis obtains expressions from the mappings $\mathrm{PDF}_{\mathrm{Dist}}$ and $\mathrm{Conditions}_{\mathrm{Dist}}$ (Section 3.2).

The rule first analyzes all of the distribution's parameters (which can represent random quantities). To iterate over the parameters, the rule uses the helper function A_e^*, defined inductively as:

$$
\begin{aligned}
A_e^*([]) &:= \lambda\sigma\,.\,(\sigma, [])\\
A_e^*(\mathrm{se} : t) &:= \lambda\sigma\,.\,\textbf{let}\ (\sigma_1, e) = A_e(\mathrm{se})(\sigma)\ \textbf{and}\ (\sigma_2, t') = A_e^*(t)(\sigma_1)\ \textbf{in}\ (\sigma_2, e : t').
\end{aligned}
$$

To ensure that distribution parameters have the correct values, the rule invokes a helper function Assert, which adds guards from the $\mathrm{Conditions}_{\mathrm{Dist}}$. Finally, the rule declares a *fresh* temporary variable τ (specified by a predicate FreshVar),

$$
\begin{aligned}
A_s:\ \text{Stmt} &\to (\Sigma \to \Sigma)\\
A_s(\textbf{skip}) &:= \lambda\sigma.\sigma\\
A_s(x := \text{se}) &:= \lambda\sigma.\,\textbf{let}\ (\sigma', e) = A_e(\text{se})(\sigma)\ \textbf{in}\ \text{Distribute}(x, \delta(x-e))(\sigma')\\
A_s(x = \text{se}) &:= A_s(x := \text{se}[\![\tau/x]\!]) \circ \text{Rename}(x, \tau),\ \text{with FreshVar}(\tau)\\
A_s(s_1; s_2) &:= A_s(s_2) \circ A_s(s_1)\\
A_s(\textbf{assert}(\text{se})) &:= \lambda\sigma.\,\textbf{let}\ (\sigma', e) = A_e(\text{se})(\sigma)\ \textbf{in}\ \text{Assert}([e \neq 0])(\sigma')\\
A_s(\textbf{observe}(\text{se})) &:= \lambda\sigma.\,\textbf{let}\ (\sigma', e) = A_e(\text{se})(\sigma)\ \textbf{in}\ \text{Observe}([e \neq 0])(\sigma')\\
A_s(\textbf{if}\ \text{se}\ \{s_1\}\ \textbf{else}\ \{s_2\}) &:= \lambda\sigma.\,\textbf{let}\ (\sigma_0, e) = A_e(\text{se})(\sigma)\\
&\qquad\ \textbf{and}\ \sigma_1 = (A_s(s_1) \circ \text{Observe}([e \neq 0]))(\sigma_0)\\
&\qquad\ \textbf{and}\ \sigma_2 = (A_s(s_2) \circ \text{Observe}([e = 0]))(\sigma_0)\\
&\qquad\ \textbf{in}\ \text{Join}(\sigma, \sigma_1, \sigma_2)\\
A_s(\textbf{return}\ (x_1, \ldots, x_n)) &:= \text{KeepOnly}(x_1, \ldots, x_n)
\end{aligned}
$$

Fig. 5. Symbolic Analysis of Statements

which is then distributed according to the density function PDF_{Dist}, using the helper function Distribute. In the definitions of Assert and Distribute, we specified the states in their expanded forms (Equation 1).

Marginalization: Marginalization aggregates the probability by summing up over the variables in an expression (e.g., local variables at the end of scope or variables in an error expression). To marginalize all variables, we define the function

$$\text{MarginalizeAll}(e[\![x_1, \ldots, x_n]\!]) := \int_{\mathbb{R}} \mathrm{d}x_1 \cdots \int_{\mathbb{R}} \mathrm{d}x_n e[\![x_1, \ldots, x_n]\!]\,.$$

The function KeepOnly performs selective marginalization. It takes as input the variables $x'_1 \ldots, x'_m$ to keep and the input state σ, and marginalizes out the remaining variables in σ's distribution expressions:

$$
\begin{aligned}
&\text{KeepOnly}(x'_1, ..., x'_m)(\lambda M.\,\textbf{case}\ M\ \textbf{of}\ (x_1, ..., x_n) \Rightarrow e_1[\![x_1, ..., x_n]\!]\,, \bot \Rightarrow e_2) =\\
&\quad \textbf{let}\ \{x''_1, ..., x''_l\} = \{x_1, ..., x_n\} \setminus \{x'_1, ..., x'_m\}\\
&\quad \textbf{in}\ \lambda M.\,\textbf{case}\ M\ \textbf{of}\ (x'_1, ..., x'_m) \Rightarrow \int_{\mathbb{R}} \mathrm{d}x''_1 \cdots \int_{\mathbb{R}} \mathrm{d}x''_l e_1[\![x_1, ..., x_n]\!]\,, \bot \Rightarrow e_2
\end{aligned}
$$

3.4 Analysis of Statements

Figure 5 presents the definition of function A_s: it analyzes each statement and produces a transformer of states: $\Sigma \to \Sigma$. The initial analysis state σ_0 is defined as follows: $\sigma_0 = (\lambda M.\,\textbf{case}\ M\ \textbf{of}\ \ \vec{x} \Rightarrow \varphi(\vec{x}), \bot \Rightarrow\ \ 0)$. Here, the function `F` under analysis has parameters $\vec{x} = (x_1, ...x_n)$ where φ is an uninterpreted function representing the joint probability density of $\vec{x}$. If `F` has no parameters, we replace $\varphi()$ with 1.

Definitions: The statement $x :=$ se declares a new variable x and distributes it as a point mass centered at e (the symbolic expression corresponding to se), i.e. the analysis binds x by multiplying the joint probability density by $\delta(x - e)$.

$$\text{Rename}(x,x')(\lambda M.\,\textbf{case}\ M\ \textbf{of}\ (x_1,...,x,...,x_n) \Rightarrow e_1[\![x_1,...,x,...,x_n]\!]\,,\bot \Rightarrow e_2) :=$$
$$\lambda M.\,\textbf{case}\ M\ \textbf{of}\ (x_1,...,x',...,x_n) \Rightarrow e_1[\![x_1,...,x',...,x_n]\!]\,,\bot \Rightarrow e_2$$

$$\text{Observe}(e[\![\vec{x}]\!])(\lambda M.\,\textbf{case}\ M\ \textbf{of}\ (\vec{x}) \Rightarrow e_1[\![\vec{x}]\!]\,,\bot \Rightarrow e_2) :=$$
$$\lambda M.\,\textbf{case}\ M\ \textbf{of}\ (\vec{x}) \Rightarrow e_1[\![\vec{x}]\!] \cdot e[\![\vec{x}]\!]\,,\bot \Rightarrow e_2$$

$$\text{Join}((\lambda M.\,\textbf{case}\ M\ \textbf{of}\ (\vec{x}) \Rightarrow e_1[\![\vec{x}]\!]\,,\bot \Rightarrow e_2),\sigma_{\text{then}},\sigma_{\text{else}}) :=$$
$$\textbf{let}\ (\lambda M.\,\textbf{case}\ M\ \textbf{of}\ (\vec{x}) \Rightarrow e_1'[\![\vec{x}]\!]\,,\bot \Rightarrow e_2') = \text{KeepOnly}(\vec{x})(\sigma_{\text{then}})$$
$$\textbf{and}\ (\lambda M.\,\textbf{case}\ M\ \textbf{of}\ (\vec{x}) \Rightarrow e_1''[\![\vec{x}]\!]\,,\bot \Rightarrow e_2'') = \text{KeepOnly}(\vec{x})(\sigma_{\text{else}})$$
$$\textbf{in}\ \lambda M.\,\textbf{case}\ M\ \textbf{of}\ (\vec{x}) \Rightarrow e_1'[\![\vec{x}]\!] + e_1''[\![\vec{x}]\!]\,,\bot \Rightarrow e_2'[\![\vec{x}]\!] + e_2''[\![\vec{x}]\!] - e_2[\![\vec{x}]\!]$$

Fig. 6. Analysis of Statements - Helper Functions

Assignments: Analysis of assignments to existing variables ($x = \text{se}$) consistently renames these variable and introduces a new variable with the previous name. The substitution $\text{se}[\![\tau/x]\!]$ renames x to τ in the source expression se, since the variable being assigned may itself occur in se. The function $\text{Rename}(x,\tau)$ alpha-renames all occurrences of the variable x to τ in an existing state (σ) to avoid capture (Figure 6). It is necessary to rename x in se separately, because se is a source program expression, while Rename renames variables in the analysis state.

Observations: Observations are handled by a call to the helper function Observe (Figure 6), which conditions the probability distribution on the given expression being true. We do not renormalize the distribution after an observation, but only once, before reporting the final result (Section 3.5). Therefore, observations do not immediately change the error part of the distribution.

Conditionals: The analysis of conditionals first analyzes the condition, and then creates two copies of the resulting state σ_0. In one of the copies, the condition is then observed to be true, and in the other copy, the condition is observed to be false. Analysis of the 'then' and 'else' statements s_1 and s_2 in the corresponding states yields σ_1 and σ_2. Finally, σ_1 and σ_2 are joined together by marginalizing all locally scoped variables, including temporaries created during the analysis of the condition, and then adding the distribution and the error terms (Join; Figure 6). We subtract the error probability in the original state to avoid counting it twice.

3.5 Final Result and Renormalization

We obtain the final result by applying the state transformer obtained from analysis of the function body to the initial state and *renormalizing* it. We define the renormalization function as:

$$\text{Renormalize}(\lambda M.\,\textbf{case}\ M\ \textbf{of}\ (x_1,\ldots,x_n) \Rightarrow e_1[\![x_1,\ldots,x_n]\!]\,,\bot \Rightarrow e_2) :=$$
$$\textbf{let}\ e_Z = \text{MarginalizeAll}(e_1) + e_2$$
$$\textbf{in}\ \lambda M.\,\textbf{case}\ M\ \textbf{of}\ (x_1,\ldots,x_n) \Rightarrow [e_Z \neq 0] \cdot e_1[\![x_1,\ldots,x_n]\!] \cdot e_Z^{-1},$$
$$\bot \Rightarrow [e_Z \neq 0] \cdot e_2 \cdot e_Z^{-1} + [e_Z = 0]$$

The function obtains a normalization expression e_Z, such that the renormalized distribution expression of the resulting state integrates to 1. This way, PSI computes a normalized joint probability distribution for the function results that depends symbolically on the initial joint distribution of the function's arguments.

3.6 Discussion

Loop Analysis: PSI analyzes loops like `for i in [0..N){...}` (as mentioned in Section 2.1) by unrolling the loop body a constant N number of times. This approach also extends to loops where the number of iterations N is a random program variable. If N can be bounded from above by a constant N_{max}, a developer can encode the loop as

```
assert (N <= Nmax);
for i in [0..Nmax) {
   if (i < N) {  /* loop body */  }
}
```

To handle `for`-loops with unbounded random variables and general `while`-loops, a developer can select N_{max} such that the probability of error (i.e., probability that the loop runs for more than N_{max} iterations) is small enough. We anticipate that this approach can be readily automated. Related techniques such as [42] and [18] employ similar approximation techniques.

Function Call Analysis: PSI can analyze multiple functions, generating for each function $f(x_1, ..., x_n)$ the density expression of its m outputs, parameterized by the unknown distribution of the function's n inputs. The distribution of the function inputs is represented by an uninterpreted function $\varphi(x_1, ..., x_n)$ which appears as a subterm in the output density expression.

The rule for the analysis of function calls (see Appendix B) first creates temporary variables $a_1, \ldots, a_n$ for each argument of f, and variables $r_1, \ldots, r_m$ for each result returned by f. The variables $a_1, \ldots, a_n$ are then initialized by the actual parameters $e_1, \ldots, e_n$ by multiplying the density of the caller by $\prod_i \delta(a_i - e_i)$. The result variables in f's density expression are renamed to match $r_1, \ldots, r_n$, and the uninterpreted function φ within f's density expression is replaced with the new density of the caller (avoiding variable capture).

Formal Argument: A standard approach to prove that the translation from the source language to the target domain (in our case, the symbolic domain) is correct is to show that the transformation preserves semantics, as in [17]. This requires a specification of semantics for both the source language and the symbolic domain language. Below, we outline how one might approach such a formal proof using denotational semantics that map programs and distribution expressions to measure transformers.

Denotational semantics for the source language is easy to define by extending [30], but defining the measure semantics for distribution expressions is more challenging. Defining measure semantics for most expression terms in the symbolic domain is simple (e.g., the measure corresponding to a sum of terms is the sum of the measures of the terms). However, the semantics of expressions

containing Dirac deltas is less immediate, since there is no general pointwise product when Dirac delta factors have overlapping sets of free variables.

To assign semantics to a product expression with Dirac delta factors, we therefore (purely formally) integrate the expression against the indicator function of the measured set and simplify it using Dirac delta identities until no Dirac deltas are left. The resulting term can then be easily interpreted as a measure. A formal proof will also need to show that this is a well-formed definition, i.e., that all ways of eliminating Dirac deltas lead to the same measure. Once the semantics for distribution expressions has been defined, the correctness proof proceeds as a straightforward induction over the source language production rules. We consider a complete formalized proof to be an interesting future work item.

4 Symbolic Optimizations

After each step of the analysis from Section 3, PSI's symbolic engine simplifies the joint posterior distribution expressions. The algorithm of this optimization engine is a fixed point computation, which applies various symbolic transformations. We selected these transformations by their ability to optimize expressions that typically arise when analyzing probabilistic programs and that have demonstrated their efficiency for practical programs (as we discuss in Section 5). We next describe three main groups of the transformations.

4.1 Algebraic Optimizations

These optimizations implement basic algebraic identities. Some examples include removing zero-valued terms in addition expressions, removing one-valued terms in multiplication expressions, distributing exponents over products, or condensing equivalent summands and factors.

4.2 Guard Simplifications

For each term in an expression with multiple Iverson brackets and/or Dirac deltas, these optimizations analyze the constraints in the bracket factors and delta factors using sound but incomplete heuristics. PSI can then (1) remove the whole term if the constraints are inconsistent and therefore the term is always zero, (2) remove a factor if it is always satisfied, e.g., if both sides of an inequality are constants, or (3) remove a bracket factor if it is implied by other factors.

Guard Linearization: Guard linearization analyzes complex Iverson brackets and Dirac deltas with the goal to rewrite expressions in such a way that all included constraints (expressions in Iverson brackets and Dirac deltas) depend on a specified variable x in a linear way. It handles constraints that are easily recognizable as compositions of quadratic polynomials, multiplications with only one factor depending on x and integer and fractional powers (including in particular multiplicative inverses).

One aspect that requires special care is that the integral of a Dirac delta along x depends on the partial derivative of its argument in the direction of

x. For example, we have $\delta(2x) = \frac{1}{2}\delta(x)$, and in general we have $\delta(f(x)) = \sum_i \frac{\delta(x-x_i)}{|f'(x_i)|}$ for $f(x_i) = 0$, whenever $f'(x_i) \neq 0$. We ensure the last constraint by performing a case split on $f'(x) = 0$, and substituting the solutions for x into the delta expression in the "equals" case. For example, $\delta(y - x^2)$ is linearized to

$$[-y \leq 0] \cdot ([x = 0] \cdot \delta(y) + [x \neq 0] \cdot \frac{1}{2\sqrt{y}}(\delta(x - \sqrt{y}) + \delta(x + \sqrt{y}))).$$

We present the details of the guard linearization algorithm in the Appendix C.

4.3 Symbolic Integration

These optimizations replace the integration terms with equivalent terms that do not contain integration symbols. If the integrated term is a sum, the symbolic engine integrates each summand separately and pulls all successfully integrated summands out of the integral. If the integrated term is a product, the symbolic engine pulls out all factors that do not contain the integration variable before performing the integration.

Integration of Terms with Deltas: The integration engine first attempts to eliminate the integration variable with a factor that is a Dirac delta, by applying the rule $f(e) = \int_{\mathbb{R}} \mathrm{d}x f(x) \cdot \delta(x - e)$. The engine can often transform deltas that depend on the integration variable x in more complicated ways into equivalent expressions only containing x-dependent deltas of the above form, using guard linearization. This transformation is applied when evaluating the integral.

Integration of Continuous Terms: The symbolic engine integrates continuous terms (without Dirac deltas) in several steps. First, it multiplies out all terms that contain the integration variable and groups together all Iverson bracket terms in a single term. Second, it computes the lower and upper bounds of integration by analyzing the Iverson bracket term. If necessary, it first rewrites the term into an equivalent term within which all Iverson brackets specify the constraints on the integration variable in a direct fashion, using guard linearization. This is necessary as in general, a single condition inside a bracket might not be equivalent to a single lower or upper bound for the integration variable. The integration bounds are then computed as the minimum of all upper bounds and the maximum of all lower bounds, where minimum and maximum are again encoded using Iverson brackets. Third, the symbolic engine applies a number of standard rules for obtaining antiderivatives, including integration of power terms, natural logaritms and exponential functions, and integration by parts. We present the details of PSI's integration rules in the Appendix C.

5 Evaluation

This section presents an experimental evaluation of PSI and its effectiveness compared to the state-of-the-art symbolic and approximate inference techniques.

Implementation: We implemented PSI using the D programming language. PSI can produce resulting query expressions in several formats including Matlab, Maple, and Mathematica. Our system and additional documentation, including the Appendix, is available at: **http://www.psisolver.org**.

Table 1. Comparison of Exact and Interactive Symbolic Inference Approaches.

Benchmark	Type	Dataset	PSI	Mathem.	Maple	Hakaru
`BurglarAlarm`	D	–	●	●	●	●
`ClinicalTrial1`	DC	100/1000	●	–	–	××
`CoinBias`	DC	5/5	●	t/o	t/o	●^{n}
`DigitRecognition`	D	784/784	●	–	–	×
`Grass`	D	–	●	●	××	●
`HIV`	C0	10/369	◐^{n}	–	–	–
`LinearRegression1`	C0	100/1000	◐$^\int$	–	–	–
`NoisyOr`	D	–	●	●	●	●
`SurveyUnbias`	DC	5/5	●^{n}	t/o	×	●^{n}
`TrueSkill`	C	3/3	◐$^\int$	t/o	t/o	◐^{n}
`TwoCoins`	D	–	●	●	●	–
`AddFun/max`	C	–	●	○	×	○
`AddFun/sum`	C	–	●	●	●	◐^{n}
`BayesPointMachine`	C	6/6	●^{n}	t/o	t/o	◐^{n}
`ClickGraph`	DC	5/5	●	t/o	t/o	●^{n}
`ClinicalTrial2`	DC	5/5	●	t/o	t/o	●^{n}
`Coins`	D	–	●	●	××	●
`Evidence/model1`	D	–	●	○	××	●
`Evidence/model2`	D	–	●	●	××	●
`LearningGaussian`	C0	100/100	◐^{n}	–	–	–
`MurderMystery`	D	–	●	●	●	●

Legend: **Type:** Discrete (D), Continuous (C), Zero-probability observations (0).
Dataset: Full (a/a), no input (–), or the first a out of b inputs (a/b).
Tools: Fully simplified (●) Partially simplified (◐), Not simplified (○), Not normalized (●^{n}, ◐^{n}), Remaining integrals (◐$^\int$), Incorrect (××), Crash (×), Timeout (t/o).

Benchmarks: We selected two sets of benchmarks distributed with existing inference engines. Specifically, we used examples from R2 [39] and Fun programs from Infer.NET 2.5 [34]. We describe these benchmarks in the Appendix D. We use the data sets and queries provided with the original computations. Out of 21 benchmarks, 10 have bounded loops. The loop sizes are usually equal to the sizes of the data sets (up to 784 data points in `DigitRecognition`). Since several benchmarks have data sets that are too large for any of the symbolic tools to successfully analyze, we report the results with truncated data sets.

5.1 Comparison with Exact Symbolic Inference Engines

Experimental Setup: For comparison with Mathematica 2015 and Maple 2015, we instruct PSI to skip symbolic integration and automatically generate distribution expressions in the formats of the two tools. We run Mathematica's `Simplify()` and Maple's `simplify()` commands. For Hakaru [38,10] (commit `e61cc72009b5cae1dee33bee26daa53c0599f0bc`), we implemented the benchmarks

as Hakaru terms in Maple, using the API exposed by the `NewSLO.mpl` simplifier (as recommended by the Hakaru developers). For each benchmark, we set a timeout of 10 minutes and manually compared the results of the tools.

Results: Table 1 presents the results of symbolic inference. For each benchmark, we present the types of variables it has and whether it has zero-probability observations. We also report the size of the data set provided by the benchmark (if applicable) and the size of the subset we used. For each tool we report the observed inference result. We mark a result as fully simplified (●) if it does not have any integrals remaining and has a small number of remaining terms. We mark results that have some integral terms remaining ($◐^{\int}$), and partially simplified results (◐). We mark a result as not normalized ($●^n$, $◐^n$) if a tool does not fully simplify the normalization constant. We marked specifically if execution of a tool experienced a crash ($\times$) a timeout (t/o) or a tool produced an incorrect result ($\times\times$). For five benchmarks, the automatic conversion of PSI's expressions could not produce Mathematica and Maple expressions, because of the complexity of the benchmarks. We marked those entries as '–'. Hakaru's simplifier does not handle zero-probability observations and expectation queries, and therefore we have not encoded these benchmarks (also marked as '–').

PSI: PSI was able to fully symbolically evaluate many of the benchmark programs and generate compact symbolic distributions. Running PSI took less than a second for most benchmarks. The most time consuming benchmark was `DigitRecognition`, which PSI analyzed in 37 seconds. For two benchmarks, PSI was not able to remove all integral terms, although it simplified and removed many intermediate integrals.

Mathematica and Maple: For several benchmarks, both Mathematica and Maple did not produce a result before the timeout, or returned a non-simplified expression as the result. This indicates that the distribution expressions of these benchmarks are too complex, causing general computer algebra systems to navigate a huge search space. However, we note that these results are obtained for a mechanized translation of programs with the specific encoding we described in this paper. It is possible that a human-driven interactive inference with an alternative encoding may result in more simplified distribution expressions.

Maple crashed for `addFun/max` and `addFun/sum`. We identified that the crashes were caused by an infinite recursion and subsequent stack overflow during simplification. Four benchmarks – `Coins`, `Evidence/model1`, `Evidence/model2`, and `Grass` produce results that are different from those produced by the other tools. For instance, Maple simplifies the density function of `Coins` to 0 (which is incorrect). We attribute this incorrectness to the way Maple integrates Dirac deltas and how it defines Heaviside functions (by default, they are undefined at input 0, but a user can provide a different setting [2]). In our evaluation, none of the alternative settings could yield the correct results. We reported these bugs to the Maple developers. These examples indicate that users should be cautious when using general computer algebra systems to analyze probabilistic programs.

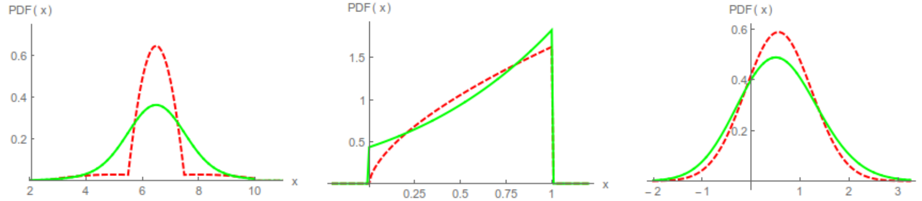

Fig. 7. `Tracking.query2`: PSI (solid; exact) and SVE (dashed).

Fig. 8. `ClickGraph`: PSI (solid; exact) and Infer.NET (dashed).

Fig. 9. `AddFun/max`: PSI (solid; exact) and Infer.NET (dashed).

Hakaru: For the `ClinicalTrial1` benchmark, Hakaru produced a result differing from PSI's. To get a reference result, we ran R2's simulation to compute an approximate result and found that this result is substantially closer to PSI's. For the `DigitRecognition` benchmark, Hakaru overflowed Maple's stack limit. Hakaru does not simplify the `AddFun/max` benchmark, but unlike Maple (which it uses), it does not crash.

Performance: Summed over all examples where Hakaru produced correct but possibly unsimplified results except `BayesPointMachine`, PSI and Hakaru ran for about the same time (8.7s and 8.8s, respectively). `BayesPointMachine` is an outlier, for which Hakaru requires 41.9s, while PSI finds a solution in 1.24s. Mathematica and Maple are 10-300 times slower than PSI. We present the detailed time measurements for each benchmark in the Appendix E.

5.2 Comparison with Approximate Symbolic Inference Engine

Experimental Setup: We compared PSI with SVE [43] by running posterior distribution queries on the models from the SVE distribution (from the commit `f4cea111f7d489933b36a43c753710bd14ef9f7f`). We included models `tracking` (with 7 provided posterior distribution queries) and `radar` (with 5 posterior distribution queries). We excluded the `competition` model because SVE crashes on it. We did not evaluate SVE on R2 and Infer.NET benchmarks as SVE does not encode some distributions (e.g., Beta or Gamma) and lacks support for Dirac deltas, significantly limiting its ability to represent assignment statements.

Results: PSI fully optimized the posterior distributions for all seven queries of the `tracking` model. PSI fully optimized one query from the `radar` benchmark and experienced timeout for the remaining queries. Figure 7 presents the posterior density functions (PDFs) for one of the `tracking` queries. SVE's polynomial approximation yields a less precise shape of the distribution compared to PSI.

5.3 Comparison with Approximate Numeric Inference Engines

Experimental Setup: We also compared the precision and performance of PSI's exact inference with the approximate inference engines Infer.NET [34] and R2 [39] for a subset of their benchmarks. Specifically, we compared PSI to Infer.NET on `ClickGraph`, `ClinicalTrial`, `AddFun/max`, `AddFun/sum`, and `MurderMystery` and compared PSI to R2 on `BurglarAlarm`, `CoinBias`, `Grass`, `NoisyOR`, and `TwoCoins`. We executed both approximate engines with their default parameters.

Results: Infer.NET produces less precise approximate distributions for `ClickGraph` and `AddFun/max` (Figures 8 and 9), Infer.NET's approximate inference is imprecise in representing the tails of the distributions, although the means of the two distributions are similar (e.g., differing by 0.7% for both benchmarks). PSI and Infer.NET produced identical distributions for the remaining benchmarks. Because of its efficient variational inference algorithms, Infer.NET computed results 5-200 times faster than PSI. The precision loss of R2 on Burglar alarm is 20% (R2's output burglary probability is 0.0036 compared to the exact probability 0.00299). For the other benchmarks, the difference between the results of PSI and R2 is less than 3%. The run times of PSI and R2 were similar, e.g., PSI was two times faster on `TwoCoins`, and R2 was two times faster on `NoisyOR`. We present details of the comparison in the Appendix F.

The examples in Figures 7, 8, and 9 illustrate that the choice of inference method depends on the context in which the inference results are used. While inferences about expectations in machine learning applications may often tolerate imprecision in return for faster or more scalable computation, many uses of probabilistic inference in domains such as security, privacy, and reliability engineering need to reason about a richer set of queries, while requiring correct and precise inference. We believe that the PSI system presented in this paper is particularly suited for such settings and is an important step forward in making automated exact inference feasible.

6 Related Work

This section discusses related work in symbolic inference and probabilistic program analysis.

6.1 Symbolic Inference

Graphical Models: Early research in the machine learning community focused on symbolic inference in Bayesian networks with discrete distributions [44] and combinations of discrete and linearly-dependent Gaussian distributions [11]. For more complex hybrid models, researchers proposed projecting distributions to mixtures of base functions, which can be easily integrated, such as truncated exponentials [36] and piecewise polynomials [45,43]. In contrast to these approximate approaches, PSI's algorithm performs exact symbolic integration.

Probabilistic Programs: Claret et al. [12] present a data flow analysis for symbolically computing exact posterior distributions for programs with discrete variables. This analysis operates on the program's concrete state, while efficiently storing the states using ADD diagrams.

Bhat et al. [6] present a type system for programs with continuous probability distributions. This approach is extended in [7] to programs with discrete and continuous variables (but only discrete observations). Like PSI, the density compiler from [7] computes posterior distribution expressions, but instead of symbolically simplifying and removing integrals, it generates a C program that performs numerical integration (which may, in general, be expensive to run).

The Hakaru probabilistic language [38,10] runs inference tasks by combining symbolic and sampling-based methods. To optimize MCMC sampling for probabilistic programs, Hakaru's symbolic optimizer (1) translates the programs to probability density expressions in Maple's language, (2) calls an extended version of Maple's simplifier, (3) uses these results to generate an optimized Hakaru program, and, if necessary, (4) calls a MCMC sampler with the optimized program. While Maple's expression language is more expressive than PSI's, it also creates a more complex search space for expression optimizations. PSI further reduces the search space by optimizing expressions after each analysis step, while in Hakaru's workflow, the distribution expression is optimized only after translating the whole program.

6.2 Probabilistic Program Analysis

Verification: Researchers presented various static analyses that verify probabilistic properties of programs, including safety, liveness, and/or expectation queries. These verification techniques have been based on abstract interpretation [35,16,14,33], axiomatic reasoning [37,29,5], model checking [26], and symbolic execution [42,18]. Many of the existing approaches compute exact probabilities of failure only for discrete distributions or make approximations when analyzing computations with both discrete and continuous distributions.

Researchers have also formalized fragments of probability theory inside general-purpose theorem provers, including reasoning about discrete [28,4] and continuous distributions [17,24]. The focus of these works is on human-guided interactive verification of (possibly recursive) programs. In contrast, PSI performs fully automated inference of hybrid discrete and continuous distributions for programs with bounded loops.

Transformation: R2 [39] transforms probabilistic programs by moving observe statements next to the sampling statement of the corresponding variable to improve performance of MCMC samplers. Gretz et al. [23] generalize this transformation to move observations arbitrarily through a program. Probabilistic program slicing [27] removes statements that are not necessary for computing a user-provided query. These transformations simplify program structure, while preserving semantics. In comparison, PSI directly transforms and simplifies the probability distribution that underlies a probabilistic program.

7 Conclusion

We presented PSI, an approach for end-to-end exact symbolic analysis of probabilistic programs with discrete and continuous variables. PSI's symbolic nature provides the necessary flexibility to answer various queries for non-trivial probabilistic programs. More precise and reliable probabilistic inference has the potential to improve the quality of the results in various application domains and help developers when testing and debugging their probabilistic models and inference algorithms. With its rich symbolic domain and optimization engine, we believe that PSI is a useful tool for studying the design of precise and scalable probabilistic inference based on symbolic reasoning.

References

1. Maple (2015), www.maplesoft.com/products/maple/
2. Maple Heaviside Function (2015), http://www.maplesoft.com/support/help/Maple/view.aspx?path=Heaviside
3. Mathematica (2015), https://www.wolfram.com/mathematica/
4. Audebaud, P., Paulin-Mohring, C.: Proofs of randomized algorithms in coq. Science of Computer Programming 74(8), 568–589 (2009)
5. Barthe, G., Köpf, B., Olmedo, F., Zanella Béguelin, S.: Probabilistic relational reasoning for differential privacy. In: ACM POPL (2012)
6. Bhat, S., Agarwal, A., Vuduc, R., Gray, A.: A type theory for probability density functions. In: ACM POPL (2012)
7. Bhat, S., Borgström, J., Gordon, A.D., Russo, C.: Deriving probability density functions from probabilistic functional programs. In: TACAS (2013)
8. Borgström, J., Gordon, A.D., Greenberg, M., Margetson, J., Van Gael, J.: Measure transformer semantics for bayesian machine learning. In: ESOP (2011)
9. Bornholt, J., Mytkowicz, T., McKinley, K.S.: Uncertain<T>: A first-order type for uncertain data. In: ASPLOS (2014)
10. Carette, J., Shan, C.c.: Simplifying probabilistic programs using computer algebra. In: PADL (2016)
11. Chang, K.C., Fung, R.: Symbolic probabilistic inference with both discrete and continuous variables. Systems, Man and Cybernetics, IEEE Transactions on 25(6), 910–916 (1995)
12. Claret, G., Rajamani, S.K., Nori, A.V., Gordon, A.D., Borgström, J.: Bayesian inference using data flow analysis. In: ESEC/FSE (2013)
13. Cooper, G.F.: The computational complexity of probabilistic inference using bayesian belief networks. Artificial intelligence 42(2), 393–405 (1990)
14. Cousot, P., Monerau, M.: Probabilistic abstract interpretation. In: ESOP (2012)
15. Dagum, P., Luby, M.: Approximating probabilistic inference in bayesian belief networks is np-hard. Artificial intelligence 60(1), 141–153 (1993)
16. Di Pierro, A., Wiklicky, H.: Probabilistic abstract interpretation and statistical testing. In: Process Algebra and Probabilistic Methods: Performance Modeling and Verification, pp. 211–212. Springer (2002)
17. Eberl, M., Hölzl, J., Nipkow, T.: A verified compiler for probability density functions. In: ESOP (2015)
18. Filieri, A., Păsăreanu, C.S., Visser, W.: Reliability analysis in symbolic pathfinder. In: ICSE (2013)
19. Gelman, A., Lee, D., Guo, J.: Stan a probabilistic programming language for bayesian inference and optimization. Journal of Educational and Behavioral Statistics (2015)
20. Gilks, W.R., Thomas, A., Spiegelhalter, D.J.: A language and program for complex bayesian modelling. The Statistician pp. 169–177 (1994)
21. Goodman, N., Mansinghka, V., Roy, D., Bonawitz, K., Tenenbaum, J.: Church: A language for generative models. In: UAI (2008)
22. Gordon, A.D., Henzinger, T.A., Nori, A.V., Rajamani, S.K.: Probabilistic programming. In: Proceedings of the on Future of Software Engineering (2014)
23. Gretz, F., Jansen, N., Kaminski, B.L., Katoen, J.P., McIver, A., Olmedo, F.: Conditioning in probabilistic programming. arXiv preprint arXiv:1504.00198 (2015)
24. Hasan, O.: Formalized Probability Theory and Applications Using Theorem Proving. IGI Global (2015)

25. Hershey, S., Bernstein, J., Bradley, B., Schweitzer, A., Stein, N., Weber, T., Vigoda, B.: Accelerating inference: towards a full language, compiler and hardware stack. arXiv preprint arXiv:1212.2991 (2012)
26. Hinton, A., Kwiatkowska, M., Norman, G., Parker, D.: Prism: A tool for automatic verification of probabilistic systems. In: TACAS (2006)
27. Hur, C.K., Nori, A.V., Rajamani, S.K., Samuel, S.: Slicing probabilistic programs. In: ACM PLDI (2014)
28. Hurd, J.: Formal verification of probabilistic algorithms. Ph.D. thesis, University of Cambridge (2001)
29. Katoen, J.P., McIver, A.K., Meinicke, L.A., Morgan, C.C.: Linear-invariant generation for probabilistic programs. In: SAS (2010)
30. Kozen, D.: Semantics of probabilistic programs. Journal of Computer and System Sciences 22(3), 328–350 (1981)
31. Kozlov, A.V., Koller, D.: Nonuniform dynamic discretization in hybrid networks. In: UAI (1997)
32. Mansinghka, V., Selsam, D., Perov, Y.: Venture: a higher-order probabilistic programming platform with programmable inference. arXiv preprint 1404.0099 (2014)
33. Mardziel, P., Magill, S., Hicks, M., Srivatsa, M.: Dynamic enforcement of knowledge-based security policies using probabilistic abstract interpretation. Journal of Computer Security 21(4), 463–532 (2013)
34. Minka, T., Winn, J., Guiver, J., Webster, S., Zaykov, Y., Yangel, B., Spengler, A., Bronskill, J.: Infer.NET 2.5 (2013), http://research.microsoft.com/infernet
35. Monniaux, D.: Abstract interpretation of probabilistic semantics. In: SAS (2000)
36. Moral, S., Rumí, R., Salmerón, A.: Mixtures of truncated exponentials in hybrid bayesian networks. In: Symbolic and Quantitative Approaches to Reasoning with Uncertainty, pp. 156–167. Springer (2001)
37. Morgan, C., McIver, A., Seidel, K.: Probabilistic predicate transformers. ACM Transactions on Programming Languages and Systems (TOPLAS) 18(3) (1996)
38. Narayanan, P., Carette, J., Romano, W., Shan, C.c., Zinkov, R.: Probabilistic Inference by Program Transformation in Hakaru (System Description)
39. Nori, A.V., Hur, C.K., Rajamani, S.K., Samuel, S.: R2: An efficient mcmc sampler for probabilistic programs. In: AAAI (2014)
40. Pfeffer, A.: Ibal: a probabilistic rational programming language. In: Proceedings of the 17th international joint conference on Artificial intelligence-Volume 1. pp. 733–740. Morgan Kaufmann Publishers Inc. (2001)
41. Sampson, A., Panchekha, P., Mytkowicz, T., McKinley, K.S., Grossman, D., Ceze, L.: Expressing and verifying probabilistic assertions. In: ACM PLDI (2014)
42. Sankaranarayanan, S., Chakarov, A., Gulwani, S.: Static analysis for probabilistic programs: inferring whole program properties from finitely many paths. In: ACM PLDI (2013)
43. Sanner, S., Abbasnejad, E.: Symbolic variable elimination for discrete and continuous graphical models. In: AAAI (2012)
44. Shachter, R.D., D'Ambrosio, B., Del Favero, B.: Symbolic probabilistic inference in belief networks. In: AAAI (1990)
45. Shenoy, P.P., West, J.C.: Inference in hybrid bayesian networks using mixtures of polynomials. International Journal of Approximate Reasoning 52(5) (2011)
46. Wood, F., van de Meent, J.W., Mansinghka, V.: A new approach to probabilistic programming inference. In: AISTATS (2014)

Appendix

A Common Probability Distributions

Table 2 presents several primitive distributions encoded in PSI's intermediate language. For a distribution Dist, the function $\mathrm{PDF}_{\mathrm{Dist}}$ returns the probability density function and the function $\mathrm{Conditions}_{\mathrm{Dist}}$ returns the conditions that the parameters should satisfy. Both inputs and the parameters can be random quantities. The translation to this intermediate language from the mathematical definition of the functions is typically straightforward.

Name (Params)	**PDF** (x, Params)	**Conditions** (x, Params)
Bernoulli (e_p)	$e_p \cdot \delta(1-x) + (1-e_p) \cdot \delta(x)$	$[0 \leq e_p] \cdot [e_p \leq 1]$
Poisson (e_λ)	$e^{-e_\lambda} \cdot \sum_{x' \in \mathbb{Z}} [0 \leq x'] \cdot \delta(x - x') \cdot e_\lambda^{x'} / \Gamma(x'+1)$	$[0 < e_\lambda]$
UniformInt (e_a, e_b)	$\frac{\sum_{x' \in \mathbb{Z}} \delta(x-x')[e_a \leq x'] \cdot [x' \leq e_b]}{\sum_{x' \in \mathbb{Z}} [e_a \leq x'] \cdot [x' \leq e_b]}$	$[\sum_{x' \in \mathbb{Z}} [e_a \leq x'] \cdot [x' \leq e_b] \neq 0]$
Uniform (e_a, e_b)	$[e_a = e_b] \cdot \delta(x - e_a) + [e_a \neq e_b] \cdot \frac{1}{(e_b - e_a)} \cdot [e_a \leq x] \cdot [x \leq e_b]$	$[e_a \leq e_b]$
Gauss (e_μ, e_ν)	$[e_\nu = 0] \cdot \delta(x - e_\mu) + [e_\nu \neq 0] \cdot \frac{\exp(-(x-e_\mu)^2/(2e_\nu))}{\sqrt{2\pi e_\nu}}$	$[0 \leq e_\nu]$
Pareto (e_a, e_b)	$e_a \cdot e_b^{e_a} \cdot x^{-(e_a+1)}$	$[0 \leq e_a] \cdot [0 \leq e_b]$
Beta (e_α, e_β)	$1/B(e_\alpha, e_\beta) \cdot x^{\alpha-1} \cdot (1-x)^{\beta-1} \cdot [0 \leq x] \cdot [x \leq 1]$	$[0 < e_\alpha] \cdot [0 < e_\beta]$
Gamma (e_α, e_β)	$\frac{\beta^\alpha}{\Gamma(\alpha)} \cdot x^{\alpha-1} \cdot \mathrm{e}^{-\beta \cdot x} \cdot [0 \leq x]$ where $\Gamma(t) := \int_0^\infty \mathrm{d}x\, x^{t-1} e^{-x}$.	$[0 < \alpha] \cdot [0 < \beta]$

Table 2. PDF and Correctness Conditions for Several Primitive Distributions.

Currently, PSI supports following distributions: `Gauss`, `Uniform`, `Exponential`, `Gamma`, `Beta`, `StudentT`, `Weibull`, `Laplace`, `Pareto`, `Rayleigh`, `Bernoulli`, `UniformInt`, and `Categorical`. Adding a new primitive distribution only requires a few lines of code.

Additionally, PSI allows the developer to specify an arbitrary PDF of the resulting distribution in PSIs intermediate language from Fig. 3 using the `SampleFrom (pdf_expression,...)` primitive.

B Symbolic Inference (Remaining Rules)

Comparison Operators The rule for translating comparison operators is as follows.

$$\begin{aligned}A_e(\mathrm{se}_1\ cop\ \mathrm{se}_2) :=& \lambda\sigma.\,\mathbf{let}\ (\sigma_1, e_1) = A_e(\mathrm{se}_1)(\sigma)\ \mathbf{and}\ (\sigma_2, e_2) = A_e(\mathrm{se}_2)(\sigma_1)\ \mathbf{in}\\ & (\sigma_2, \mathrm{TranslateCop}(cop, e_1, e_2))\end{aligned}$$

where

$cop \in \{==, \neq, <, >, \leq, \geq\}$,

$\mathrm{TranslateCop}(==, e_1, e_2) := [e_1 = e_2], \mathrm{TranslateCop}(\neq, e_1, e_2) := [e_1 \neq e_2]$,

$\mathrm{TranslateCop}(<, e_1, e_2) := [e_1 < e_2], \mathrm{TranslateCop}(>, e_1, e_2) := [e_2 < e_1]$,

$\mathrm{TranslateCop}(\leq, e_1, e_2) := [e_1 \leq e_2], \mathrm{TranslateCop}(\geq, e_1, e_2) := [e_2 \leq e_1]$.

Function Calls We describe the distribution expression transformer corresponding to some source language function f with n arguments and m return values by expressions $e_f[\![x'_1, \ldots, x'_m]\!]$ and e'_f each containing φ, which is an uninterpreted function with n arguments. The expressions describe the distribution of the results and the probability of error respectively. Such expressions can be obtained by running our analysis on the function body in the initial state

$$\begin{aligned}\sigma_0 =& (\lambda M.\,\mathbf{case}\ M\ \mathbf{of}\\ & (x_1, \ldots, x_n) \Rightarrow \varphi(x_1, \ldots, x_n)\\ & \bot \Rightarrow\ 0.\end{aligned}$$

and extracting the two distribution expressions from the final state.

The rule for translating a function call is then given by

$$\begin{aligned}A_e(f(\mathrm{se}_1, ..., \mathrm{se}_n)) :=& \lambda\sigma.\,\mathbf{let}\ \mathrm{FreshVars}(a_1, \ldots, a_n)\ \mathbf{and}\ \mathrm{FreshVars}(r_1, \ldots, r_m)\\ & \mathbf{and}\ e''_f[\![r_1, \ldots, r_m]\!] := e_f[\![r_1/x'_1]\!] \cdots [\![r_m/x'_m]\!]\\ & \mathbf{and}\ \sigma' := (A_s(a_n := \mathrm{se}_n) \circ \cdots \circ A_s(a_1 := \mathrm{se}_1))(\sigma)\\ & \mathbf{and}\ \sigma'' := \lambda M.\,\mathbf{case}\ M\ \mathbf{of}\\ & (x_1, \ldots, x_k, r_1, \ldots, r_m) \Rightarrow\\ & \quad e''_f[\![(\lambda(a_1, \ldots, a_n).\,e_c[\![x_1, \ldots, x_k, a_1, \ldots, a_n]\!])/\varphi]\!]\,,\\ & \bot \Rightarrow\ e'_c + \int_{\mathbb{R}^k} \mathrm{d}x_1 ... \mathrm{d}x_k e'_f[\![(\lambda(a_1, ..., a_n).\,e_c[\![x_1, ..., x_k, a_1, ..., a_n]\!])/\varphi]\!]\\ & \mathbf{in}\ (\sigma'', (r_1, \ldots, r_m))\end{aligned}$$

where $e_c[\![x_1, \ldots, x_k, a_1, \ldots, a_n]\!]$ and e'_c are expressions with

$$\begin{aligned}\sigma' = \lambda M.\,\mathbf{case}\ M\ \mathbf{of}\ & (x_1, \ldots, x_k, a_1, \ldots, a_n) \Rightarrow\\ & e_c[\![x_1, ..., x_k, a_1, ..., a_n]\!]\\ & \bot \Rightarrow\ e'_c\end{aligned}$$

C Rules for Symbolic Integration

In this section, we give a more detailed exposition of the rules we use to replace the integration terms with equivalent terms that do not contain integration symbols.

If the integrated term is a sum, the symbolic engine integrates each summand separately and pulls all successfully integrated summands out of the integral. If the integrated term is a product, the symbolic engine pulls out all factors that do not contain the integration variable before performing the integration.

Guard Normalization Internally, all brackets are represented using only the forms $[e = 0], [e \neq 0]$ and $[e \leq 0]$. If Iverson brackets occur nested within Dirac deltas or other Iverson brackets, these enclosing deltas or brackets are replaced by an exhaustive case distinction on all possible combinations of Iverson bracket conditions, pruning unsatisfiable conditions if possible. For example, the expression $[[x = 0] \cdot [y \leq 0] = 0]$ is transformed into $[x = 0] \cdot [y \leq 0] \cdot [1 \cdot 1 = 0] + [x \neq 0] \cdot [y \leq 0] \cdot [0 \cdot 1 = 0] + [x = 0] \cdot [-y \leq 0] \cdot [y \neq 0] \cdot [1 \cdot 0 = 0] + [x \neq 0] \cdot [-y \leq 0] \cdot [y \neq 0] \cdot [0 \cdot 0 = 0]$ which is in turn simplified to $[x \neq 0] \cdot [y \leq 0] + [x = 0] \cdot [-y \leq 0] \cdot [y \neq 0] + [x \neq 0] \cdot [-y \leq 0] \cdot [y \neq 0]$.

Guard Linearization To transform an expression into an equivalent expression that only contains guards depending linearly on a given variable x, all subexpressions that are Iverson brackets or Dirac deltas are rewritten. In fact, our algorithm will isolate the variable x from all other variables in case it succeeds (i.e., the variable is an isolated summand in the constraints it occurs in). Guard linearization is a sound but incomplete heuristic that works for many practically relevant constraints; it does not solve arbitrary non-linear constraints (this is an undecidable problem). We handle constraints that are easily recognizable as compositions of quadratic polynomials, multiplications with only one factor depending on x and integer and fractional powers (including in particular multiplicative inverses).

For the following, we assume that Dirac delta constraint expressions have only countably many zeros. Due to guard normalization, we can assume that all such expressions are differentiable at all points where they are defined.

Note that this discussion is not entirely formal; expressions are sometimes used interchangeably with their meaning. Furthermore, we treat additional assumptions that hold for the duration of a recursive call implicitly.

For a given Iverson bracket or Dirac delta, first, the constraint expression (i.e. e in $\delta(e)$ or $[e \leq 0], [e = 0], [e \neq 0]$) is normalized by distributing products over sums containing the variable x. We then use a recursive algorithm to linearize the constraint. The recursion state contains four variables p, l and r; each of those contains an expression from our distribution expression language. The recursive algorithm maintains different invariants for the different kinds of guards:

- Iverson brackets of the form $[e \leq 0]$:
  $[e \leq 0]$ is equivalent to $[0 \leq p] \cdot [l \leq r] + [p < 0] \cdot [r \leq l]$.

- Iverson brackets of the form $[e = 0]$:
  $[e = 0]$ is equivalent to $[l = r]$.
- Dirac deltas $\delta(e)$:
  $\delta(e)$ is equivalent to $\sum_{x' \in \mathbb{R}: l[\![x'/x]\!]=r, (\frac{\partial}{\partial x} e)[\![x'/x]\!] \neq 0} \frac{\delta(x-x')}{(\frac{\partial}{\partial x} e)[\![x'/x]\!]} + [\frac{\partial}{\partial x} e = 0] \cdot \delta(e)$.

Furthermore, throughout the execution of the recursive algorithm, x does not occur free in r. p is ignored for guards that are not of the shape $[e \leq 0]$.

The invariants, together with the information which type of Iverson bracket/Dirac delta is being handled allow computation of a guard which is equivalent to the original guard at any point in the recursive algorithm (except for Dirac deltas, where we additionally need to be able to solve the equation $l = r$). Intuitively, the recursive algorithm recurses, maintaining the invariant, until the original guard can be either easily written in a linear shape given the invariant, or it can be determined that linearization is not supported.

We start the recursion with $p = 1, l = e$ and $r = 0$ such that the invariants are satisfied.

One invocation of the recursive algorithm performs the following case distinction on the shape of l:

- Case $l = l_1 + \cdots + l_n$:
  The summands are partitioned according to whether x occurs free in them. In case only one summand l_i has this property, the algorithm recurses with $p' = p, l' = l_i$ and $r' = r - l_i$. Otherwise, if $l - r$ can be written as a quadratic polynomial $a \cdot x^2 + b \cdot x + c$, the algorithm performs a symbolic case split on $a = 0$: It recurses with $l' = b \cdot x + c, r' = r$ and multiplies the result by $[a = 0]$. Additionally, it creates symbolic expressions for the discriminant $d = b^2 - 4ac$ and the two roots $z_1 = (-b - \sqrt{d})/(2 \cdot a), z_2 = (-b + \sqrt{d})/(2 \cdot a)$. From those expressions, an appropriate expression handling the nondegenerate quadratic can be built using moderately complex expressions with Iverson brackets/Dirac deltas depending at most linearly on x. (The related case of an arbitrary even power is discussed below in more detail. This should clarify how this expression is built.) This expression is computed (for $[e \neq 0]$, $[e = 0]$ and $\delta(e)$, this involves additional recursive calls with $l' = x, r' = z_{1,2}$), multiplied by $[a = 0]$ and added to the expression handling the degenerate case.
- Case $l = l_1 \cdots l_n$:
  The summands are partitioned according to whether x occurs free in them. In case only one summand l_i has this property, the algorithm performs a symbolic case split on $0 = L := l_1 \cdots l_{i-1} \cdot l_{i+1} \cdots l_n$ (by encoding both branches symbolically and multiplying them with the appropriate Iverson brackets $[L = 0]$ and $[L \neq 0]$ respectively, both not depending on x): For $L = 0$, the condition reduces to an Iverson bracket/Dirac delta of the original kind with an expression of $-r$. As $-r$ does not depend on x, such guards have no non-linear dependencies on x. For $L \neq 0$, the algorithm recurses with $p' = p \cdot L, l' = l_i$ and $r' = r/L$. Linearization is not performed for (non-polynomial) constraints with more than one x-dependent factor.

- Case $l = l_1^{l_2}, 0 > l_2$ constant :
  The algorithm computes the expression $i = l_1^{-l_2}$ and performs a symbolic case split on $r = 0$. For $r \neq 0$, the algorithm recurses with $p' = -p \cdot r \cdot i$, $l' = l'$,$r' = r^{-1}$. For $r = 0$, the result depends on the kind of guard under consideration. For $[e \leq 0]$, guard linearization is applied recursively to the expression $[p \cdot i \leq 0]$. For $[e \neq 0]$, the result is given by 1 and for $[e = 0]$ it is 0. (This exploits the fact that the inverse of a real number is never zero.)
- Case $l = l_1^n, n$ even integer :
  The algorithm first computes symbolic expressions for the two roots $z_2 = r^{1/n}, z_1 = -z_2$. The result depends on the kind of guard under consideration.
  - For $[e \leq 0]$: Constraint linearization is performed recursively on $[0 \leq p]$. The algorithm performs a symbolic case split on the result. For $0 \leq p$, the algorithm recurses with $p' = -1, l' = l_1, r' = z_1$ and $p'' = 1, l'' = l_1, r' = z_2$, multiplies the results (in order to get an appropriate expression describing the constraint that l_1 is between the two roots) and multiplies with a factor $[r \geq 0]$ (in order to ensure that the subexpressions z_1 and z_2 are well-formed). For $p < 0$, the algorithm recurses with $p' = 1, l' = l_1, r' = z_1$ and $p'' = -1, l'' = l_1, r'' = z_2$ adds the results (in order to get an appropriate expression describing the constraint that l_1 is not strictly inbetween the two roots), after multiplying the second summand with $[z_2 \neq 0]$ (in order to avoid double-counting a root at 0) and multiplies with a factor $[0 < r]$ (in order to ensure that the subexpressions z_1 and z_2 are well-formed) finally, the expression $[r \leq 0]$ is added as a summand (as all even powers are at least 0).
  - For $[e \neq 0]$ and $[e = 0]$, the algorithm recurses with $l' = l_1, r' = z_1$ and $l'' = l_1, r'' = z_2$, multiplies/adds (avoiding double-counting) the results respectively, and multiplies with a factor $[0 \leq r]$. For $[e \neq 0]$, the algorithm finally adds a summand $[r < 0]$
- Case $l = l_1^n, n$ odd integer:
  The algorithm performs a symbolic case split on $0 \leq r$. For $0 \leq r$, the algorithm recurses with $p' = p$, $l' = l_1$ and $r' = r^{1/n}$. For $r < 0$, the algorithm recurses with $p' = p$, $l' = l_1$ and $r' = -(-r)^{1/n}$ (this is necessary because the simplifier never considers power expressions with negative base and fractional exponent well-formed).
- Case $l = l_1^{m/n}, m, n$ integers:
  The algorithm recurses on $p' = p, l' = l_1, r' = r^{n/m}$ and multiplies the result with $[0 \leq r]$ (in order to make sure $r^{n/m}$ is well-formed). For $[e \leq 0]$, the algorithm additionally adds a summand $p' \cdot [r < 0]$ where p' is the result of running constraint linearization recusively on $[p < 0]$. For $[e < 0]$, the algorithm additionally adds a summand $[r < 0]$.
- Case $l = x$:
  For Iverson brackets, the algorithm can simply return the precise expressions which are known to be equivalent to the original guards due to the invariants, as those already have the required shape.
  For Dirac deltas, the sum simplifies, as the only possible value for x' is r, which does not depend on x, and hence the remaining Dirac delta has

a constraint expression of the right shape. The only term in the invariant which is potentially problematic is $[\frac{\partial}{\partial x} e = 0] \cdot \delta(e)$. Intuitively, our approach will be to solve the equation $\frac{\partial}{\partial x} e = 0$ for x. Then we can substitute x for its solution within $\delta(e)$, which makes the Delta completely independent of x. The algorithm simply calls constraint linearization recursively on $[\frac{\partial}{\partial x} e = 0]$ and distributes products over sums and simplifies the result. Then the algorithm iterates over all summands of the resulting expression. For each summand s, the algorithm iterates over all factors and tries to find an Iverson bracket of the shape $[x - e' = 0]$. The algorithm then computes $s \cdot \delta(e[\![e'/x]\!])$. All of these expressions are then added together to form the expression e''. In case the algorithm is not able to determine a suitable e' for some summand, guard linearization fails. The result is given by linearize$([\frac{\partial}{\partial x} e = 0]) \cdot \frac{\delta(x-r)}{\frac{\partial}{\partial x} e} + e''$. (Where "linearize" performs guard linearization recursively.)
- Otherwise: If l is not given in any of the discussed shapes, guard linearization fails.

Continuous Integration If no Dirac delta or uninterpreted function is found in the integrand, the integral to be evaluated is continuous. In this case, the symbolic engine first linearizes Iverson bracket constraints in the integration variable. If all Iverson brackets are successfully linearized, upper bounds and lower bounds on the integration variable can be extracted by solving a simple linear equation for each such Iverson bracket. The engine computes symbolic expressions for the minimum of all upper bounds and the maximum of all lower bounds. (Using the encoding $\min(a, b) = [a \leq b] \cdot a + [b < a] \cdot b$.) Note that the bounds of integration can also be $-\infty$ and ∞ respectively.

Once integration bounds have been determined, it suffices to compute an antiderivative for the remaining factors of the integrand, by the fundamental theorem of calculus. The symbolic engine applies a number of standard rules for symbolic integration in order to compute antiderivatives:

- Powers of the integration variable without free integration variable in the exponent (including the special cases $x = x^1$ and $1 = x^0$) are handled by the standard rule

$$\int \mathrm{d}x\, x^y = [y + 1 \neq 0] \cdot \frac{x^{y+1}}{y+1} + [y + 1 = 0] \cdot \log|x| + C$$

- Powers that only contain the integration variable in the exponent are handled by rewriting them from x^y to $\mathrm{e}^{y \cdot \log|x|}$. If the exponent $z(x) := y \cdot \log|x|$ is a linear function in x, PSI applies the standard rule

$$\int \mathrm{d}x e^{z(x)} = \frac{e^{z(x)}}{z'(x)} + C.$$

- The natural logarithm is integrated as

$$\int \mathrm{d}x \log(a \cdot x + b) = a^{-1} \cdot (a \cdot x + b) \cdot \log(a \cdot x + b) - x + C.$$

- For $a > 0$, $\int \mathrm{d}x\, \mathrm{e}^{-a \cdot x^2 + b \cdot x + c} = \mathrm{e}^{\frac{b^2}{4a} + c} \frac{1}{\sqrt{a}} \cdot (\mathrm{d}/\mathrm{d}x)^{-1} [e^{-x^2}] \left(\sqrt{a} \cdot x - \frac{b}{2 \cdot \sqrt{a}}\right) + C$
- The antiderivative of $\log(x)^y / x$ is given by

$$[y + 1 = 0] \cdot \log(|\log(|x|)|) + [y + 1 \neq 0] \cdot \frac{\log(|x|)^{y+1}}{y + 1} + C.$$

- For $\Gamma(a, z) = \int_{\mathbb{R}} \mathrm{d}t [z \leq t] \cdot t^{a-1} \cdot \mathrm{e}^{-t}$ and n a positive integer constant, $\int \mathrm{d}x \log(a \cdot x + b)^n = \frac{(-1)^n}{a} \cdot \Gamma(1 + n, -\log(a \cdot x + b)) + C$.
- For $a > 0$, $\int \mathrm{d}x\, (\mathrm{d}/\mathrm{d}x)^{-1} [e^{-x^2}](a \cdot x + b) =$

$$\frac{1}{a} (\mathrm{d}/\mathrm{d}x)^{-1} [e^{-x^2}](a \cdot x + b) \cdot (a \cdot x + b) - \mathrm{e}^{-(a \cdot x + b)^2} + C$$

- The antiderivative of a Gaussian times its own antiderivative is evaluated via partial integration.
- If the integrand has the form $x^n \cdot f(x)$ for some positive integer n, and the symbolic evaluation engine is able to find an antiderivative $F(x)$ for $f(x)$ as well as for $n \cdot x^{n-1} \cdot F(x)$, the antiderivative for the integrand is computed via partial integration as

$$\int \mathrm{d}x\, x^n \cdot f(x) = x^n \cdot F(x) + \int \mathrm{d}x\, n \cdot x^{n-1} F(x).$$

  All expressions encountered in this fashion are tracked, and if a linear relation is discovered for an antiderivative, it is computed by solving the corresponding linear equation.

The antiderivatives are then evaluated at the computed bounds. This possibly necessitates evaluating a limit in case one or more of the bounds is infinite. We implemented a number of standard rules to evaluate limits and remove them from the final distribution expression.

Sums and Products of Terms If the integrated term is a sum, the symbolic integration engine attempts to integrate each summand separately and pulls all successfully integrated summands out of the integral. If the integrated term is a product, the symbolic integration engine pulls out all factors that do not contain the integration variable before continuing the integration.

Evaluating Limits We implemented a number of simple rules to evaluate limits. For example, for a sum, some summands will have a finite limit, some summands will go to ∞ and some summands will go to $-\infty$. To compute a limit, more case splits may become necessary. For example $\lim_{x \to \infty} e^{-a \cdot x^2}$ where a does not depend on x is 1 if a is zero, ∞ if a is negative and 0 if a is positive. The necessary case splits are performed using Iverson brackets.

In case the engine is able to evaluate all necessary limits and they are finite in all cases, the final result is then the product of an Iverson bracket checking that the lower bound is at most the upper bound times the difference of the antiderivative evaluated at the upper and lower bound respectively.

D Benchmarks

We selected programs from those that are distributed with the existing R2 [39] and Infer.NET [34] inference systems. R2 benchmarks include:

- **BurglarAlarm**: Finds the probability of burglary, given that the alarm sounded.
- **ClinicalTrial1**: Find if a new medical treatment is effective given the observations of its outcome on the experimental and control groups.
- **CoinBias**: Finds the bias of a biased coin given the toss observations.
- **DigitalRecognition**: Recognizes numerical digits from handwritten notes.
- **Grass**: Finds the probability of raining, given that the grass is wet.
- **HIV**: Estimates the parameters of a linear HIV dynamical model from data.
- **LinearRegression1**: Computes a best fit line given data observations.
- **NoisyOR**: Finds the posterior distribution of a node's value (computed as the noisy-or function of the values of the node's parents) in a directed acyclic graph.
- **SurveyUnbias**: Computes a gender bias for a survey report, models population with a Gaussian distribution.
- **TrueSkill**: Computes the skills of players in a series of games, given the outcomes of these games.
- **TwoCoins**: Finds the marginal distributions, two fair coins, given that not both tosses resulted in heads.

Infer.NET Fun language benchmarks include:

- **AddFun/Max**: Computes a maximum of Gaussian variables.
- **AddFun/Sum**: Computes a sum of Gaussian variables, with a filter.
- **BayesPointMachine**: Training a Bayes point machine.
- **ClickGraph**: Finds the relevance of a web page from the sequence of a user's clicks.
- **ClinicalTrial2**: Find if a new medical treatment is effective given the observations of its outcome on the experimental and control groups. Differs from the R2 version in the parameters and the query.
- **Coins**: Two coins example. The query is the full joint posterior distribution.
- **Evidence/Model1**: Tossing a single coin, with a prior evidence that influences whether the coin is tossed.
- **Evidence/Model2**: Tossing two coins, with a a prior evidence that determines whether two or one coins are tossed.
- **LearningGaussian**: Learning mean and variance of a Gaussian distribution from a set of data points.
- **MurderMystery**: Finds the probability that a person is the murderer given the weapon.

Table 3. Comparison of Exact and Interactive Symbolic Inference Approaches.

Benchmark	PSI	Mathematica	Maple	Hakaru
`BurglarAlarm`	0.07	4.57	4.06	0.02
`ClinicalTrial1`	17.15	–	–	××0.54
`CoinBias`	0.10	t/o	t/o	0.02
`DigitRecognition`	36.97	–	–	×6.24
`Grass`	0.68	7.89	2.00	0.02
`HIV`	1.14	–	–	–
`LinearRegression1`	2.93	–	–	–
`NoisyOr`	0.53	118.89	57.72	0.04
`SurveyUnbias`	1.22	t/o	×585.09	0.23
`TrueSkill`	0.38	t/o	t/o	5.36
`TwoCoins`	0.04	9.77	0.57	0.01
`AddFun/max`	0.05	31.66	×2.02	0.126
`AddFun/sum`	0.03	9.77	3.89	0.213
`BayesPointMachine`	1.24	t/o	t/o	41.95
`ClickGraph`	3.56	t/o	t/o	2.01
`ClinicalTrial2`	0.87	t/o	t/o	0.72
`Coins`	0.01	0.58	××0.46	0.01
`Evidence/model1`	0.01	0.55	××0.44	0.01
`Evidence/model2`	0.01	6.40	××0.42	0.01
`LearningGaussian`	15.01	–	–	–
`MurderMystery`	0.02	0.93	0.51	0.01

We elided the benchmarks **LDA** and **MixtureOfGaussians**, which use Dirichlet distributions, which we and Hakaru do not currently support.

E Comparison with Exact Symbolic Approaches

Table 3 presents the timing results for each benchmark and the tool. Columns 2-5 present time in seconds required to compute the output distribution of each benchmark. We specifically marked if a benchmark caused a tool to timeout (t/o), crash (×) or produce a wrong result (××). We ran the timing experiment on an Intel(R) i7 CPU at 3.40GHz, with 16 GB of RAM, running Linux OS.

F Comparison with Approximate Inference Approaches

Tables 4 and 5 present the comparison of PSI with R2 and Infer.NET approximate inference engines for a subset of the benchmarks. For each comparison we present columns that represent the name of the benchmark, the results (in the symbolical domain form) that PSI computed for the probabilistic query, the result of the alternative approach (R2 and Infer.NET), and the the inference times of PSI and alternative approaches. We collected inference times of these approaches from tool diagnostics (and removed the time to set up their analysis).

Table 4. R2 Benchmark Precision and Analysis Time

Benchmark	Expectation PSI	Expectation R2	Time PSI	Time R2
BurglarAlarm	$2969983/992160802 \approx 0.002993\ldots$	0.0036	70 ms	173 ms
CoinBias	$5/12 = 0.41\bar{6}$	0.417	100 ms	356 ms
Grass	$509/719 \approx 0.7079\ldots$	0.715	680 ms	336 ms
NoisyOR	$130307/160000 \approx 0.8144\ldots$	0.814	530 ms	250 ms
TwoCoins	$(1/3, 1/3)$	$(0.324, 0.336)$	40 ms	108 ms

Table 5. Infer.NET Benchmark Precision and Analysis Time.

Benchmark	Distribution PSI	Distribution Infer.NET
BurglarAlarm	$(2969983 \cdot \delta(1-b) + 989190819 \cdot \delta(b))/992160802$	Bernoulli(0.002995)
ClickGraph	$\frac{6(s+3)^5}{3367} \cdot [0 \leq s] \cdot [s \leq 1]$	Beta(1.625, 1)
ClinicalTrial	$900/102 \cdot c \cdot (1-c)^4 \cdot t^4 \cdot (1-t) \cdot$ $(77 \cdot \delta(1-e) + 25 \cdot \delta(e))$	(Beta(5,2), Beta(2,5)) Bernoulli(0.755))
AddFun/max	$\sqrt{2}/\pi \cdot (d/dx)^{-1}[e^{-x^2}](r/\sqrt{2}) \cdot e^{-r^2/2}$	Gauss(0.56, 0.68)
AddFun/sum	$(2 \cdot \sqrt{\pi})^{-1} \cdot \mathrm{e}^{-r^2/4}$	Gauss(0, 2)
MurderMystery	$(9 \cdot \delta(1-p) + 560 \cdot \delta(p))/569$	Bernoulli(0.0158)

Benchmark	Time PSI	Time Infer.NET
BurglarAlarm	70 ms	14 ms
ClickGraph	3.56 s	20 ms
ClinicalTrial	870 ms	21 ms
AddFun/max	50 ms	4 ms
AddFun/sum	30 ms	4 ms
MurderMystery	20 ms	1 ms

Precision of Expectation Queries For all benchmarks in Table 4, PSI produces precise expected values (and does not need to apply approximations). The precision loss of R2 on Burglar alarm is 20%. For the other benchmarks, this difference is smaller than 3%. For TwoCoins, R2 is not able to find equal expectations even though the two outputs have the same marginal distribution. With an exception of NoisyOr, both tools have similar inference times.

Precision of Posterior Distribution Queries For all benchmarks in Table 5, PSI produces precise results PSI was able to compute precise closed-form expressions for all output distributions. For three examples (BurglarAlarm, GaussSum, and MurderMystery) Infer.NET computes the almost the same distributions (with numerical error of distribution parameters less than 0.01%). For ClinicalTrial, Infer.NET produces the distributions of two independent variables. Table 5 contains the joint distribution from which these marginals can be immediately derived. Infer.NET produces less precise approximate distributions for ClickGraph and AddFun/max.