SPPL: Probabilistic Programming with Fast Exact Symbolic Inference

Feras A. Saad
Massachusetts Institute of Technology
Cambridge, MA, USA

Martin C. Rinard
Massachusetts Institute of Technology
Cambridge, MA, USA

Vikash K. Mansinghka
Massachusetts Institute of Technology
Cambridge, MA, USA

Abstract

We present the Sum-Product Probabilistic Language (Sppl), a new probabilistic programming language that automatically delivers exact solutions to a broad range of probabilistic inference queries. Sppl translates probabilistic programs into *sum-product expressions*, a new symbolic representation and associated semantic domain that extends standard sum-product networks to support mixed-type distributions, numeric transformations, logical formulas, and pointwise and set-valued constraints. We formalize Sppl via a novel translation strategy from probabilistic programs to sum-product expressions and give sound exact algorithms for conditioning on and computing probabilities of events. Sppl imposes a collection of restrictions on probabilistic programs to ensure they can be translated into sum-product expressions, which allow the system to leverage new techniques for improving the scalability of translation and inference by automatically exploiting probabilistic structure. We implement a prototype of Sppl with a modular architecture and evaluate it on benchmarks the system targets, showing that it obtains up to 3500x speedups over state-of-the-art symbolic systems on tasks such as verifying the fairness of decision tree classifiers, smoothing hidden Markov models, conditioning transformed random variables, and computing rare event probabilities.

CCS Concepts: • **Mathematics of computing** → *Probabilistic representations*; *Probabilistic inference problems*; • **Software and its engineering** → *Formal language definitions*.

Keywords: probabilistic programming, symbolic execution

ACM Reference Format:
Feras A. Saad, Martin C. Rinard, and Vikash K. Mansinghka. 2021. SPPL: Probabilistic Programming with Fast Exact Symbolic Inference. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3453483.3454078

PLDI '21, June 20–25, 2021, Virtual, Canada

ACM ISBN 978-1-4503-8391-2/21/06.
https://doi.org/10.1145/3453483.3454078

1 Introduction

Reasoning under uncertainty is a well-established theme across diverse fields including robotics, cognitive science, natural language processing, algorithmic fairness, amongst many others [14, 21, 31, 60]. A common approach for modeling uncertainty is to use probabilistic programming languages (PPLs [29]) to both represent complex probability distributions and perform probabilistic inference within the language. There is growing recognition of the utility of PPLs for solving challenging tasks that involve probabilistic reasoning in various application domains [8, 27, 34, 35].

Probabilistic inference is central to reasoning about uncertainty and is a central concern for both PPL implementors and users. Several PPLs leverage approximate inference techniques [28, 59, 67], which have been used effectively in a variety of settings [11, 17, 55]. Drawbacks of approximate inference, however, include a lack of accuracy and/or soundness guarantees [18, 38]; difficulties with programs that combine continuous, discrete, or mixed-type distributions [11, 68]; challenges assessing the quality of iterative solvers [9]; and the substantial expertise needed to write custom inference programs that deliver acceptable performance [17, 39]. To address the shortcomings of approximate inference, several PPLs instead use exact symbolic techniques [6, 10, 23, 43, 69]. These languages can typically express a large class of models, using general computer algebra to solve queries. However, the generality of the symbolic computations causes them to sometimes fail, even on problems with tractable solutions.

Our Work We introduce the Sum-Product Probabilistic Language (Sppl), a system that occupies a new point in the expressiveness vs. performance trade-off space for exact symbolic inference. A key idea in Sppl is to incorporate certain modeling restrictions that avoid the need for general computer algebra, instead using a new, specialized class of "sum-product" symbolic expressions to exactly represent probability distributions specified by Sppl programs. These new symbolic expressions extend and generalize sum-product networks [47], which are computational graphs that have received widespread attention for their clear probabilistic semantics and tractable properties for exact inference—see [64] for a comprehensive and curated literature review. These sum-product expressions are used to automatically obtain exact solutions to probabilistic inference queries about Sppl programs, which are fast and scalable in tractable regimes.

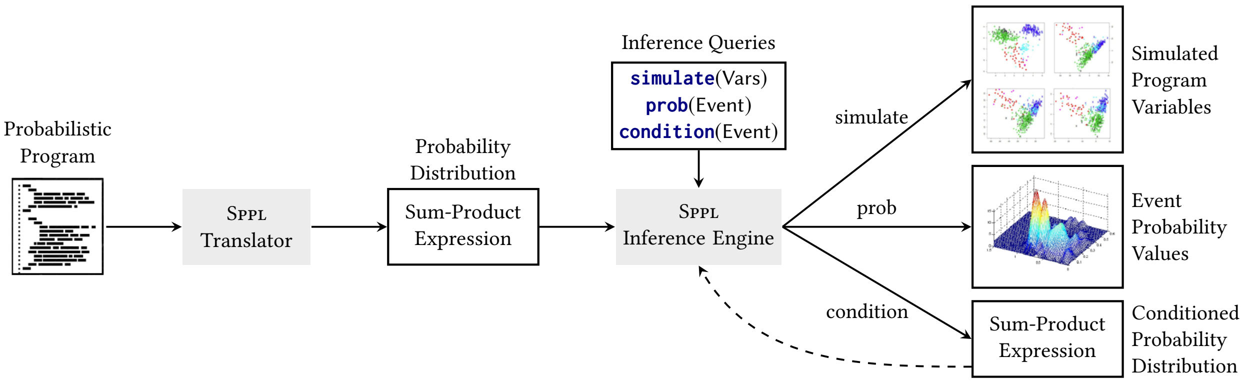


Figure 1. Sppl system overview. Probabilistic programs are translated into symbolic sum-product expressions that represent the joint distribution over all program variables and are used to deliver exact solutions to probabilistic inference queries.

System Overview Fig. 1 shows an overview of our approach. Given a probabilistic program written in Sppl (Lst. 2) a translator (Lst. 3) produces a sum-product expression that represents the prior distribution over all program variables. Given this expression and a query specified by the user, the Sppl inference engine returns an exact answer, where:

1. **simulate**(Vars) returns random samples of program variables from their joint probability distribution;
2. **prob**(Event) returns the probability of an event, which is a predicate on program variables;
3. **condition**(Event) returns a new sum-product expression for the posterior distribution over program variables, given that the specified event is true.

A key aspect of the system design in Fig. 1 is modularity: modeling, conditioning, and querying are factored into distinct stages that reflect the essential components of a Bayesian workflow. Moreover, the dashed back-edge in Fig. 1 indicates that the new sum-product expression returned by **condition** can be reused to interactively invoke additional queries on the posterior distribution. This closure property enables substantial runtime gains across multiple datasets and queries.

Trade-offs Sppl imposes restrictions on probabilistic programs that specifically rule out the following constructs: (i) unbounded loops; (ii) multivariate numeric transformations; and (iii) arbitrary prior distributions on continuous parameters. As a result, Sppl is not designed to express model classes such as regression with a prior on real coefficients; neural networks; support-vector machines; spatial Poisson processes; urn processes; and hidden Markov models with unknown transition matrices. The aforesaid model classes cannot be represented as sum-product expressions, and most of them do not have tractable algorithms for exact inference.

We impose these restrictions to ensure that valid Sppl programs can always be translated into finite sum-product expressions, as opposed to general symbolic algebra expressions. The resulting sum-product expressions delivered by Sppl have a number of characteristics that make them a particularly useful translation target for probabilistic programs:

- **Completeness and Decomposibility:** By satisfying important completeness and decomposability conditions from the literature [47, Defs. 4,5], sum-product expressions are guaranteed to represent normalized probability distributions.
- **Efficient Factorization:** By specifying multivariate probability distributions compositionally in terms of sums and products of simpler distributions, sum-product expressions can be simplified by algebraic "factorization" (Fig. 3d, Fig. 6a).
- **Efficient Deduplication:** When an Sppl program specifies a generative model with conditional independence structure, the translated sum-product expression typically contains identical subexpressions that can be "deduplicated" into a single logical node in memory (Fig. 3d, Fig. 6b).
- **Efficient Caching:** Inference algorithms for sum-product expressions proceed from root to leaves to root, allowing intermediate results to be cached and reused at deduplicated internal subexpressions in a depth-first graph traversal.
- **Closure Under Conditioning:** Sum-product expressions are closed under probabilistic conditioning (Thm. 4.1), which allows them to be reused across multiple datasets and inference queries about the same probabilistic program.
- **Linear-Time Exact Inferences:** For a well-defined class of common queries, inference scales linearly in the expression size (Thm. 4.3); when Sppl delivers a "small" expression after factorization and deduplication, inference is also fast.

It is well-known that a very large class of tractable models can be cast as sum-product networks [47, Thm. 2]. Sppl automatically constructs these representations from generative probabilistic programs that use standard constructs such as arrays, if/else branches, for-loops, and numeric and logical operators. To enable this translation, Sppl introduces new sum-product expressions and inference algorithms that extend standard sum-product networks by supporting (many-to-one) univariate transformations, mixed-type base measures, and pointwise and set-valued constraints. These constructs make Sppl expressive enough to solve prominent inference tasks in the PPL literature [2, 36, 46, 68] for which standard sum-product networks have not been previously used. Example model classes include most finite discrete models, latent variable models with discrete hidden states and arbitrary observed states, and decision trees over discrete and continuous variables. Taken together, these characteristics make Sppl particularly effective for fast and scalable inference on tractable problems, with low variance runtime and complete, usable answers to users. Our experimental evaluation (Sec. 6) indicates that Sppl delivers these benefits on the problems it is designed to solve, whereas more general and expressive techniques in previous solvers [2, 4, 23] typically exhibit orders of magnitude worse performance on these problems, runtime has higher variance, and/or results may be unusable, i.e., with unsimplified symbolic integrals.

Key contributions We identify the following contributions:

- **New semantic domain for sum-product expressions** (Sec. 3) that extends sum-product networks [47] by including mixed-type distributions, numeric transforms, logical formulas, and events with pointwise and set-valued constraints.
- **Provably sound exact symbolic inference algorithms** (Sec. 4) based on a proof that sum-product expressions are closed under conditioning on any event that can be specified in the domain. We use these algorithms to build an efficient and multi-stage inference architecture that separates model translation, conditioning, and querying into distinct stages, enabling interactive workflows and computation reuse.
- **The Sum-Product Probabilistic Language** (Sec. 5), a PPL built on a novel translation semantics from generative code to sum-product expressions, which are used to deliver exact inferences to queries. We present optimization techniques to improve scalability of translation and inference by exploiting conditional independences and repeated structure.
- **Empirical measurements of efficacy** (Sec. 6) on inference tasks from the literature that Sppl targets, which show that it delivers substantial improvements over existing baselines, including up to 3500x speedup over state-of-the-art fairness verifiers [2, 4] and symbolic integration [23], as well as many orders of magnitude speedup over sampling-based inference [40] for computing rare event probabilities.

2 Overview

We next describe two examples that illustrate the programming style in Sppl and queries supported by the system.

2.1 Indian GPA Problem

The Indian GPA problem is a canonical example that has been widely considered in the probabilistic programming literature [44, 45, 48, 57, 68] for its use of a "mixed-type" random variable that takes both continuous and discrete values, depending on the random branch taken by the program.

Specifying the Prior Fig. 2a shows the generative process for three variables (`Nationality`, `Perfect` and `GPA`) of a student. The student's nationality is either India or USA with equal probability (line 1). Students from India (line 2) have a 10% probability of a perfect 10 GPA (lines 3-4), otherwise the GPA is uniform over $[0, 10]$ (line 5). Students from USA (line 6) have a 15% probability of a perfect 4 GPA (lines 6-7), otherwise the GPA is uniform over $[0, 4]$ (line 8).

Prior Sum-Product Expression The graph in Fig. 2d shows the translated sum-product expression for this program, which represents a sampler for the distribution over program variables as follows: (i) if a node is a sum (+), visit a random child with probability equal to the weight of the edge pointing to the child; (ii) if a node is a product (×), visit each child exactly once, in no specific order; (iii) if a node is a leaf, sample a value from the distribution at the leaf and assign it to the variable at the leaf. Similarly, the graph encodes the joint distribution of the variables by treating (i) each sum node as a probabilistic mixture; (ii) each product node as a tuple of independent variables; and (iii) each leaf node as a primitive random variable. Thus, the prior distribution is:

$$\begin{aligned}
&\Pr[\texttt{Nationality} = n, \texttt{Perfect} = p, \texttt{GPA} \le g] \qquad (1)\\
&= 0.5\big[\delta_{\text{India}}(n) \cdot (0.1[(\delta_{\text{True}}(p) \cdot \mathbf{1}[10 \le g])]\\
&\quad + 0.9[(\delta_{\text{False}}(p) \cdot (g/10 \cdot \mathbf{1}[0 \le g < 10] + \mathbf{1}[10 \le g]))])\big]\\
&+ 0.5\big[\delta_{\text{USA}}(n) \cdot (0.15[(\delta_{\text{True}}(p) \cdot \mathbf{1}[4 \le g])]\\
&\quad + 0.85[(\delta_{\text{False}}(p) \cdot (g/4 \cdot \mathbf{1}[0 \le g < 4] + \mathbf{1}[4 \le g]))])\big].
\end{aligned}$$

Fig. 2b shows Sppl queries for the prior marginal distributions of the three variables, plotted in Fig. 2e. The two jumps in the cumulative distribution function (CDF[1]) of `GPA` at 4 and 10 correspond to the atoms that occur when `Perfect` is true. The piecewise linear behavior on $[0, 4]$ and $[4, 10]$ follows from the conditional uniform distributions of `GPA`.

Conditioning the Program Fig. 2f shows an example of the **`condition`** query, which specifies an event e on which to constrain executions of the program. An event is a predicate on (possibly transformed) program variables that can be used for both **`condition`** (Fig. 2f) and **`prob`** (Fig. 2c). Sppl is the first system with inference algorithms for sum-product expressions that handle predicates of this form. Given e, the object of inference is the full posterior distribution:

$$\begin{aligned}
&\Pr[\texttt{Nationality} = n, \texttt{Perfect} = p, \texttt{GPA} \le g \mid e] \qquad (2)\\
&:= \Pr[\texttt{Nationality} = n, \texttt{Perfect} = p, \texttt{GPA} \le g, e]/\Pr[e].
\end{aligned}$$

Posterior Sum-Product Expression Given the prior expression (Fig. 2d) and conditioning event e (Fig. 2f), Sppl produces a new expression (Fig. 2g) that specifies a distribution which is precisely equal to Eq. (2), From Thm. 4.1, conditioning an Sppl program on any event that can be specified in the language results in a posterior distribution that also admits an exact sum-product expression. Conditioning on e performs several transformations on the prior expression, which are:

[1] For a real-valued random variable X, the cumulative distribution function $F : \text{Real} \to [0, 1]$ is given by $F(r) := \Pr[X \le r]$.

```
Nationality ~ choice({'India': 0.5, 'USA': 0.5})
if (Nationality == 'India'):
    Perfect ~ bernoulli(p=0.10)
    if Perfect:         GPA ~ atom(10)
    else:               GPA ~ uniform(0, 10)
else: # Nationality is 'USA'
    Perfect ~ bernoulli(p=0.15)
    if Perfect:         GPA ~ atom(4)
    else:               GPA ~ uniform(0, 4)
```

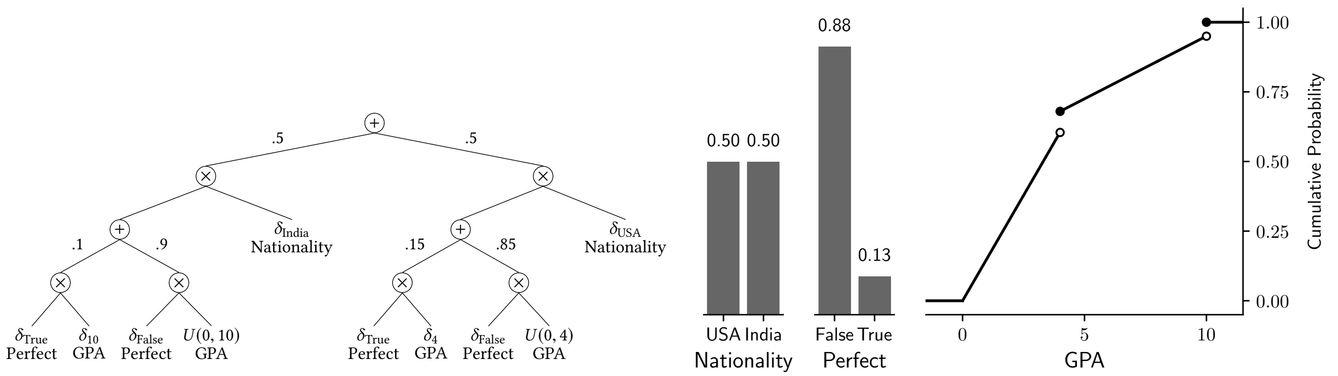
(a) Probabilistic Program

```
prob (Nationality == 'USA');
prob (Perfect == 1);
prob (GPA <= x/10) # for x = 0, ..., 120
```

(b) Example Queries on Marginal Probabilities

```
prob ((Perfect == 1)
  or (Nationality == 'India') and (GPA > 3))
```

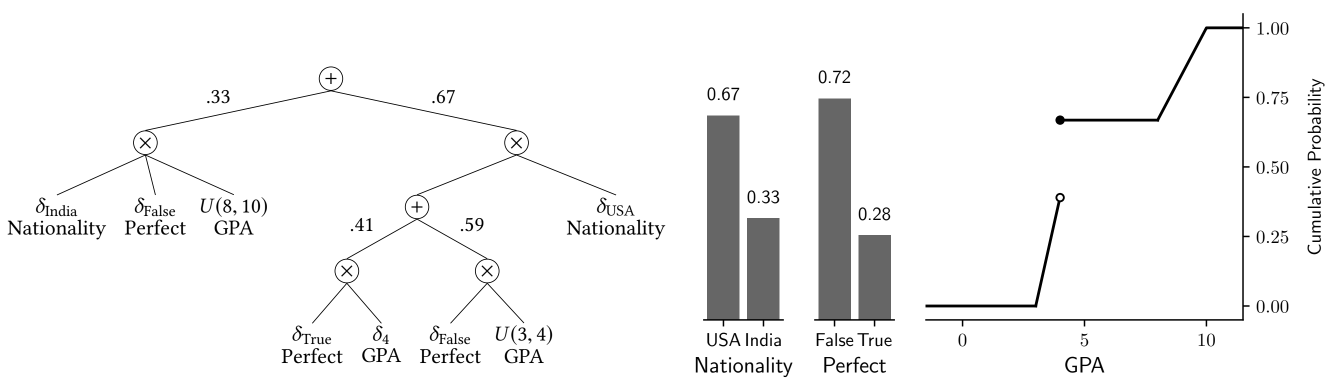
(c) Example Query on Joint Probabilities



(d) Prior Sum-Product Expression **(e)** Prior Marginal Distributions

```
condition ((Nationality == 'USA') and (GPA > 3)) or (8 < GPA < 10)
```

(f) Conditioning the Program



(g) Posterior Sum-Product Expression **(h)** Posterior Marginal Distributions

Figure 2. Analyzing the Indian GPA problem in Sppl.

1. Eliminating the subtree rooted at the parent of leaf δ_{10}, which is inconsistent with the conditioning event.
2. Rescaling the distribution $U(0, 10)$ at the leaf node in the India subtree to $U(8, 10)$.
3. Rescaling the distribution $U(0, 4)$ at the leaf node in the USA subtree to $U(3, 4)$.
4. Reweighting the branch probabilities of the sum node in the USA subtree from [.15, .85] to [.41, .59], where $.41 = .15/(.15 + .2125)$ is the posterior probability of $(\texttt{Perfect} = 1, \texttt{GPA} = 4)$ given the condition e.
5. Reweighting the branch probabilities at the root from [.5, .5] to [.33, .67] (same rules as in the previous item).

Fig. 2g shows the posterior expression obtained by applying these transformations. Using this expression, the right-hand side of Eq. (2), which is the object of inference, is

$$\Pr[\texttt{Nationality} = n, \texttt{Perfect} = p, \texttt{GPA} \le g \mid e] \tag{3}$$
$$= .33\big[\delta_{\text{India}}(n) \cdot \delta_{\text{False}}(p) \cdot (\tfrac{g-8}{2} \cdot \mathbf{1}[8 \le g < 10] + \mathbf{1}[10 \le g])\big]$$
$$+ .67\big[\delta_{\text{USA}}(n) \cdot (.41[(\delta_{\text{True}}(p) \cdot \mathbf{1}[4 \le g])]$$
$$+ .59[(\delta_{\text{False}}(p) \cdot (\tfrac{g}{4} \cdot \mathbf{1}[0 \le g < 4] + \mathbf{1}[4 \le g]))])\big].$$

(Floats are shown to two decimal places.) We can now run the **prob** queries in Fig. 2b on the conditioned program to plot the posterior marginal distributions, which are shown in Fig. 2h. The example in Fig. 2 illustrates a typical modular workflow in Sppl (Fig. 1), where modeling (Fig. 2a), conditioning (Fig. 2f) and querying (Figs. 2b–2c) are separated into distinct and reusable stages that together express the main components of Bayesian modeling and inference.

2.2 Scalable Inference in a Hierarchical HMM

The next example shows how to perform efficient smoothing in a hierarchical hidden Markov model (HMM [42]) and illustrates the optimization techniques used by the Sppl translator (Sec. 5.1), which exploit conditional independence to ensure that the size of the sum-product expression grows linearly in the number of timesteps. The code box in Fig. 3a shows a hierarchical hidden Markov model with Bernoulli hidden states Z_t and Normal–Poisson observations (X_t, Y_t). The `separated` variable indicates whether the mean values of X_t and Y_t at $Z_t = 0$ and $Z_t = 1$ are well-separated. The `p_transition` vector specifies that the current state Z_t switches from the previous state Z_{t-1} with 20% probability. This example leverages the Sppl **array**, **for**, and **switch-cases** statements, where the latter is a macro that expands to **if-else** statements (as in, e.g., the C language):

$$\textbf{switch}\ x\ \textbf{cases}\ (x'\,\textbf{in}\ \mathit{values})\ \{C\} \tag{4}$$

$$\overset{\text{desugar}}{\rightsquigarrow}\ \textbf{if}\ (x\ \textbf{in}\ \mathit{values}[0])\ \{C[x'/\mathit{values}[0]]\}$$
$$\textbf{elif}\ \ldots$$
$$\textbf{elif}\ (x\ \textbf{in}\ \mathit{values}[n{-}1])\ \{C[x'/\mathit{values}[n-1]]\},$$

where n is the length of *values* and $C[x/E]$ indicates syntactic replacement of x with expression E in command C.

The top and middle plots in Fig. 3b show a realization of X and Y that result from simulating the process for 100 time steps. The blue and orange regions along the x-axes indicate whether the true hidden state Z is 0 or 1, respectively (these "ground-truth" values of Z are not observed but need to be inferred from X and Y). The bottom plot in Fig. 3b shows the exact posterior marginal probabilities $\Pr[Z_t = 1 \mid x_{0:99}, y_{0:99}]$ for each $t = 0, \ldots, 99$ as inferred by Sppl (an inference known as "smoothing"). These probabilities track the true hidden state, i.e., the posterior probabilities that $Z_t = 1$ are low in the blue and high in the orange regions.

Fig. 3c shows a "naive" sum-product expression for the distribution of all program variables up to the first two time steps. This expression is a sum-of-products, where the products in the second level are an enumeration of all possible realizations of program variables, so that the number of terms scales exponentially in the number of time steps. Fig. 3d shows the expression constructed by Sppl, which is (conceptually) based on factoring and sharing common terms in the two level sum-of-products in Fig. 3c. These factorizations and deduplications exploit conditional independences and repeated structure in the program (Sec. 5.1), which here delivers a expression whose size scales linearly in the number of time points. Sppl can also efficiently solve variants of smoothing, e.g., computing posterior marginals $\Pr[Z_t \mid x_{0:t}, y_{0:t}]$ (filtering) or the posterior joint $\Pr[Z_{0:t} \mid x_{0:t}, y_{0:t}]$ for any t.

3 A Core Calculus for Sum-Product Expressions

This section presents a semantic domain of sum-product expressions that generalizes sum-product networks [47] and enables precise reasoning about them. This domain will be used to (i) establish the closure of sum-product expressions under conditioning on events expressible in the calculus (Thm. 4.1); (ii) describe sound algorithms for exact Bayesian inference in our system (Appx. D); and (iii) describe a procedure for translating a probabilistic program into a sum-product expression in the core language (Sec. 5). Lst. 1 shows denotations of the key syntactic elements (Lst. 9 in Appx. A) in the calculus, which includes real and nominal outcomes (Lst. 1a); real transforms (Lst. 1b); predicates with pointwise and set-valued constraints (Lst. 1c); primitive distributions (Lst. 1e); and multivariate distributions specified compositionally as sums and products of primitive distributions (Lst. 1f).

Basic Outcomes Random variables in the calculus take values in the Outcome := Real + String domain. The symbol + here indicates a sum (disjoint-union) data type, whose elements are formed by the injection operation, e.g., $\downarrow^{\text{Real}}_{\text{Outcome}} r$ for $r \in$ Real. This domain is used to model mixed-type random variables, such as X in the following Sppl program:

```
Z ~ normal(0, 1)
if   (Z <= 0):     X ~ "negative" # string
elif (0 < Z < 4):  X ~ 2*exp(Z)   # continuous real
elif (4 <= Z):     X ~ atomic(4)  # discrete real
```

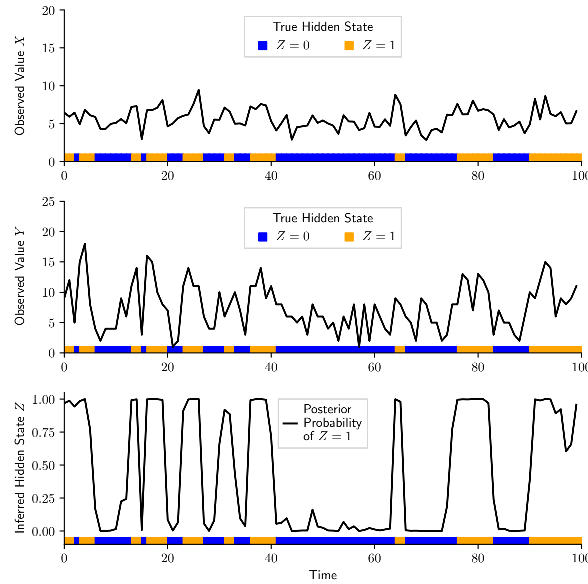
The Outcomes domain denotes a subset of Outcome as defined by the valuation function $\mathbb{V}$ (Lst. 1a). For example, $((b_1\, r_1)\ (r_2\, b_2))$ specifies a (open, closed, or clopen) real interval and $\{s_1 \ldots s_m\}^b$ is a set of strings, where b = #t indicates the complement (meta-variables such as m indicate an arbitrary but finite number of repetitions of a domain variable or subexpression). The operations *union*, *intersection*, and *complement* operate on Outcomes in the usual way (while preserving certain semantic invariants, see Appx. B)

A Sigma Algebra of Outcomes To speak precisely about random variables and measures on Outcome, we define a sigma-algebra $\mathcal{B}(\text{Outcome}) \subset \mathcal{P}(\text{Outcome})$ as follows:

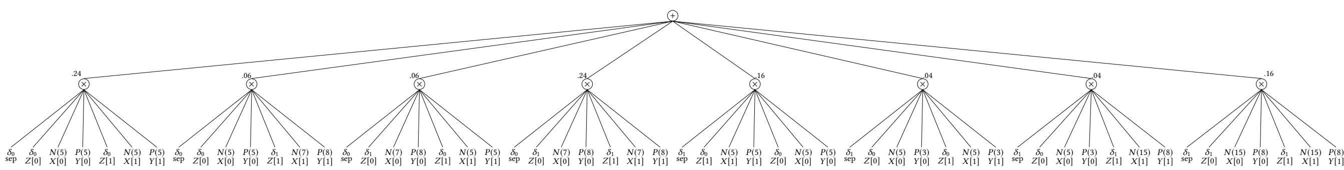
1. Let τ_{Real} be the usual topology on Real.
2. Let τ_{String} be the discrete topology on String.
3. Let $\tau_{\text{Outcome}} := \tau_{\text{Real}} \uplus \tau_{\text{String}}$ be the disjoint-union topology on Outcome, where U is open iff $\{r \mid (\downarrow^{\text{Real}}_{\text{Outcome}} r) \in U\}$ is open in Real and $\{s \mid (\downarrow^{\text{String}}_{\text{Outcome}} s) \in U\}$ is open in String.
4. Let $\mathcal{B}(\text{Outcome})$ be the Borel sigma-algebra of τ_{Outcome}.

```
p_transition = [.2, .8]
mu_x = [[5, 7], [5, 15]]
mu_y = [[5, 8], [3, 8]]

n_step = 100
Z = array(n_step)
X = array(n_step)
Y = array(n_step)

separated ~ bernoulli(p=.4)
switch separated cases (s in [0,1]):
  Z[0] ~ bernoulli(p=.5)
  switch Z[0] cases (z in [0, 1]):
    X[0] ~ normal(mu_x[s][z], 1)
    Y[0] ~ poisson(mu_y[s][z])
  for t in range(1, n_step):
    switch Z[t-1] cases (z in [0, 1]):
      Z[t] ~ bernoulli(p_transition[z])
        switch Z[t] cases (z in [0, 1]):
          X[t] ~ normal(mu_x[s][z], 1)
          Y[t] ~ poisson(mu_y[s][z])
```

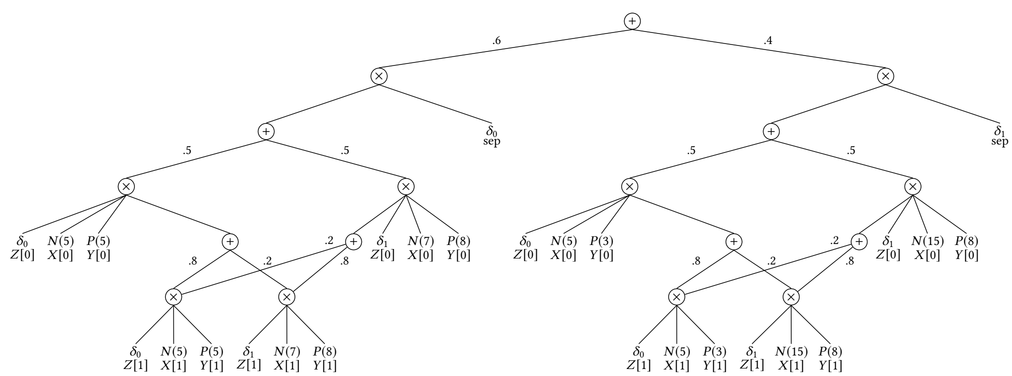
(a) Probabilistic Program



(b) Observed Data X, Y and Inferred Hidden States Z



(c) Naive Sum-Product Expression (Scales Exponentially)



(d) Optimized Sum-Product Expression (Scales Linearly)

Figure 3. Fast smoothing in a hierarchical hidden Markov model using Sppl by constructing an efficient sum-product expression that exploits conditional independences in the generative process. Optimization techniques are discussed in Sec. 5.1.

Remark 3.1. As measures on Real are defined by their values on open intervals and measures on String on singletons, we can speak of probability measures on $\mathcal{B}(\text{Outcome})$ as mappings from Outcomes to $[0, 1]$.

Real Transformations Lst. 1b shows real transformations that can be applied to variables in the calculus. The Identity Transform, written $\text{Id}(x)$, is a terminal subexpression of any Transform t and contains a single variable name that specifies the "dimension" over which t is defined. The list of all transforms is in Appx. C.1. The key operation involving transforms is computing the preimage of Outcomes v under t using *preimg* : Transform $\to$ Outcomes $\to$ Outcomes which satisfies the following properties:

$$(\downarrow^{\text{Real}}_{\text{Outcome}} r) \in \mathbb{V}\,[\![\textit{preimg}\ t\ v]\!] \iff \mathbb{T}\,[\![t]\!]\,(r) \in \mathbb{V}\,[\![v]\!]$$

$$(\downarrow^{\text{String}}_{\text{Outcome}} s) \in \mathbb{V}\,[\![\textit{preimg}\ t\ v]\!] \iff (t \in \text{Identity}) \land (s \in \mathbb{V}\,[\![v]\!]).$$

Appx. C.2 presents a symbolic solver that implements *preimg* for each Transform, which is leveraged to enable exact probabilistic inferences on transformed variables in Sppl. Fig. 4 and Appx. C.3 show example inferences with transforms.

Outcome $\coloneqq$ Real + String
$\mathbb{V}$: Outcomes $\to \mathcal{P}$(Outcome)
$\mathbb{V}[\![\varnothing]\!] \coloneqq \varnothing$ [Empty]
$\mathbb{V}\left[\!\!\left[\{s_1 \dots s_m\}^b\right]\!\!\right] \coloneqq \textbf{if } b \textbf{ then } \cup_{i=1}^m \{(\downarrow {}^{\text{String}}_{\text{Outcome}}\, s_i)\}$ [FiniteStr]
$\qquad \textbf{else } \{(\downarrow {}^{\text{String}}_{\text{Outcome}}\, s) \mid \forall i. s \neq s_i\}$
$\mathbb{V}[\![\{r_1 \dots r_m\}]\!] \coloneqq \cup_{i=1}^m \{(\downarrow {}^{\text{Real}}_{\text{Outcome}}\, r_i)\}$ [FiniteReal]
$\mathbb{V}[\![((b_1\, r_1)\ (r_2\, b_2))]\!] \coloneqq \{(\downarrow {}^{\text{Real}}_{\text{Outcome}}\, r) \mid r_1 <_{b_1} r <_{b_2} r_2\}$ [Interval]
$\qquad$ where $<_{\#t} \coloneqq <;\ <_{\#f} \coloneqq \leq;\ r_1 < r_2$
$\mathbb{V}[\![v_1 \amalg \cdots \amalg v_m]\!] \coloneqq \cup_{i=1}^m \mathbb{V}[\![v_i]\!]$ [Union]

(a) Outcomes

$\mathbb{T}$: Transform $\to$ Real $\to$ Real
$\mathbb{T}[\![\texttt{Id}(x)]\!] \coloneqq \lambda r'.r'$; $\mathbb{T}[\![\texttt{Reciprocal}(t)]\!] \coloneqq \lambda r'.1/(\mathbb{T}[\![t]\!](r'))$;
$\mathbb{T}[\![\texttt{Abs}(t)]\!] \coloneqq \lambda r'.|\mathbb{T}[\![t]\!](r')|$; $\mathbb{T}[\![\texttt{Root}(t\ n)]\!] \coloneqq \lambda r'. \sqrt[n]{\mathbb{T}[\![t]\!](r')}$;
$\mathbb{T}[\![\texttt{Poly}(t\ r_0\ \dots\ r_m)]\!] \coloneqq \lambda r'. \sum_{i=0}^m r_i (\mathbb{T}[\![t]\!](r'))^i; \dots$

(b) Transformations (Lst. 17 in Appx. C.1)

$\mathbb{E}$: Event $\to$ Var $\to$ Outcomes
$\mathbb{E}[\![(t \texttt{ in } v)]\!]\, x \coloneqq \textbf{if } (\textit{vars}\, t) = \{x\} \textbf{ then } (\textit{preimg}\ t\ v) \textbf{ else } \varnothing$ [Contains]
$\mathbb{E}[\![e_1 \sqcap \cdots \sqcap e_m]\!]\, x \coloneqq (\textit{intersection}\ \mathbb{E}[\![e_1]\!]\, x\ \dots\ \mathbb{E}[\![e_m]\!]\, x)$ [Conjunction]
$\mathbb{E}[\![e_1 \sqcup \cdots \sqcup e_m]\!]\, x \coloneqq (\textit{union}\ \mathbb{E}[\![e_1]\!]\, x\ \dots\ \mathbb{E}[\![e_m]\!]\, x)$ [Disjunction]

(c) Events

$\mathbb{P}_0[\![S]\!]$: SPE $\to$ Event $\to$ Natural $\times [0, \infty)$
$\mathbb{P}_0[\![\texttt{Leaf}(x\ d\ \sigma)]\!]\ (\texttt{Id}(x) \texttt{ in} \{rs\}) \coloneqq \textbf{match } d$ [Leaf]
$\quad \triangleright \texttt{DistR}(F\ r_1\ r_2) \Rightarrow \textbf{match } rs$
$\qquad \triangleright r \Rightarrow (1, \mathbf{1}[r_1 \leq r \leq r_2]\, F'(r)/[F(r_2) - F(r_1)])$
$\qquad \triangleright s \Rightarrow (1, 0)$
$\quad \triangleright \textbf{else} \Rightarrow \textbf{let } w \textbf{ be } \mathbb{P}[\![\texttt{Leaf}(x\ d\ \sigma)]\!]\ (\texttt{Id}(x) \texttt{ in} \{rs\}) \textbf{ in } (\mathbf{1}[w=0], w)$
$\mathbb{P}_0[\![(S_1\ w_1) \oplus \cdots \oplus (S_m\ w_m)]\!]\ \sqcap_{i=1}^\ell (\texttt{Id}(x_i) \texttt{ in} \{rs_i\}) \coloneqq$
$\quad \textbf{let}_{1 \leq i \leq m} (d_i, p_i) \textbf{ be } \mathbb{P}_0[\![S_i]\!] \left(\sqcap_{i=1}^\ell (\texttt{Id}(x_i) \texttt{ in} \{rs_i\})\right)$ [Sum]
$\quad \textbf{in if } \forall_{1 \leq i \leq m}.\ p_i = 0 \textbf{ then } (1, 0)$
$\qquad \textbf{else let } d^* \textbf{ be } \min\{d_i \mid 1 \leq i \leq m, 0 < p_i\}$
$\qquad\quad \textbf{in } (d^*, \sum_{i=1}^m \mathbf{1}[d_i = d^*]\, w_i p_i)$
$\mathbb{P}_0[\![S_1 \otimes \cdots \otimes S_m]\!]\ \sqcap_{i=1}^\ell (\texttt{Id}(x_i) \texttt{ in} \{rs_i\}) \coloneqq$ [Product]
$\quad \textbf{let}_{1 \leq i \leq m} (d_i, p_i) \textbf{ be match } \{x_1, \dots, x_m\} \cap (\textit{scope}\, S_i)$
$\qquad \triangleright \{n_1, \dots, n_k\} \Rightarrow \mathbb{P}_0[\![S_i]\!]\ \sqcap_{t=1}^k (\texttt{Id}(x_{n_t}) \texttt{ in} \{rs_t\})$
$\qquad \triangleright \{\} \Rightarrow (0, 1)$
$\quad \textbf{in } (\sum_{i=1}^n d_i, \prod_{i=1}^m p_i)$

(d) Sum-Product Expressions (Density Semantics)

$\mathbb{D}$: Distribution $\to$ Outcomes $\to [0, 1]$
$\mathbb{D}\left[\!\!\left[\texttt{DistS}((s_i\ w_i)_{i=1}^m)\right]\!\!\right] v \coloneqq$ [DistStr]
$\textbf{match } (\textit{intersection}\ v\ \{s_1 \dots s_m\}^{\#f})$
$\quad \triangleright \varnothing \mid \{r'_1 \dots r'_m\} \mid ((b_1\ r_1)\ (r_2\ b_2)) \Rightarrow 0$
$\quad \triangleright v_1 \amalg \cdots \amalg v_m \Rightarrow \sum_{i=1}^m \mathbb{D}\left[\!\!\left[\texttt{DistS}((s_i\ w_i)_{i=1}^m)\right]\!\!\right] v_i$
$\quad \triangleright \{s'_1\ \dots\ s'_k\}^b \Rightarrow \textbf{let } w \textbf{ be } \sum_{i=1}^m (w_i \textbf{ if } s_i \in \{s'_j\}_{j=1}^k \textbf{ else } 0)$
$\qquad \textbf{in if } \bar{b} \textbf{ then } w \textbf{ else } 1 - w$
$\mathbb{D}[\![\texttt{DistR}(F\ r_1\ r_2)]\!]\, v \coloneqq \textbf{match } (\textit{intersection}\ ((\#f\ r_1)\ (r_2\ \#f))\ v)$ [DistReal]
$\quad \triangleright \varnothing \mid \{r'_1 \dots r'_m\} \mid \{s'_1\ \dots\ s'_k\}^b \Rightarrow 0$
$\quad \triangleright v_1 \amalg \cdots \amalg v_m \Rightarrow \sum_{i=1}^m \mathbb{D}[\![\texttt{DistR}(F\ r_1\ r_2)]\!]\, v_i$
$\quad \triangleright ((b'_1\ r'_1)\ (r'_2\ b'_2)) \Rightarrow \dfrac{F(r'_2) - F(r'_1)}{F(r_2) - F(r_1)}$
$\mathbb{D}[\![\texttt{DistI}(F\ r_1\ r_2)]\!]\, v \coloneqq \textbf{match } (\textit{intersection}\ ((\#f\ r_1)\ (r_2\ \#f))\ v)$ [DistInt]
$\quad \triangleright \varnothing \mid \{s'_1\ \dots\ s'_k\}^b \Rightarrow 0$
$\quad \triangleright v_1 \amalg \cdots \amalg v_m \Rightarrow \sum_{i=1}^m \mathbb{D}[\![\texttt{DistI}(F\ r_1\ r_2)]\!]\, v_i$
$\quad \triangleright \{r'_1 \dots r'_m\} \Rightarrow \dfrac{\sum_{i=1}^m \begin{bmatrix} \textbf{if } (r'_i = \lfloor r'_i \rfloor) \land (r_1 \leq r'_i \leq r_2) \\ \textbf{then } F(r') - F(r'-1) \textbf{ else } 0 \end{bmatrix}}{F(\lfloor r_2 \rfloor) - F(\lceil r_1 \rceil - 1)}$
$\quad \triangleright ((b'_1\ r'_1)\ (r'_2\ b'_2)) \Rightarrow \textbf{let } \tilde{r}_1 \textbf{ be } \lfloor r'_1 \rfloor - \mathbf{1}\left[(r'_1 = \lfloor r'_1 \rfloor) \land \bar{b}'_1\right]$
$\qquad \textbf{in let } \tilde{r}_2 \textbf{ be } \lfloor r'_2 \rfloor - \mathbf{1}\left[(r'_2 = \lfloor r'_2 \rfloor) \land \bar{b}'_2\right]$
$\qquad \textbf{in } \dfrac{F(\tilde{r}_2) - F(\tilde{r}_1)}{F(\lfloor r_2 \rfloor) - F(\lceil r_1 \rceil - 1)}$

(e) Primitive Distributions

$\mathbb{P}$: SPE $\to$ Event $\to [0, 1]$
$\mathbb{P}[\![\texttt{Leaf}(x\ d\ \sigma)]\!]\, e \coloneqq \mathbb{D}[\![d]\!]\,(\mathbb{E}[\![(\textit{subsenv}\ e\ \sigma)]\!]\, x)$ [Leaf]
$\mathbb{P}[\![(S_1\ w_1) \oplus \cdots \oplus (S_m\ w_m)]\!]\, e \coloneqq \textbf{let } Z \textbf{ be } \sum_{i=i}^m w_i$ [Sum]
$\qquad \textbf{in } \sum_{i=1}^m (\mathbb{P}[\![S_i]\!]\, e) w_i / Z$
$\mathbb{P}[\![S_1 \otimes \cdots \otimes S_m]\!]\, e \coloneqq \textbf{match } (\textit{dnf}\ e)$ [Product]
$\quad \triangleright (t \texttt{ in } v) \Rightarrow \textbf{let } n \textbf{ be } \min\{1 \leq i \leq m \mid (\textit{vars}\, e) \subset (\textit{scope}\, S_i)\}$
$\qquad \textbf{in } \mathbb{P}[\![S_n]\!]\, e$
$\quad \triangleright (e_1 \sqcap \cdots \sqcap e_\ell) \Rightarrow$
$$\prod_{1 \leq i \leq m} \begin{bmatrix} \textbf{match } \{1 \leq j \leq \ell \mid (\textit{vars}\, e_j) \subset (\textit{scope}\, S_i)\} \\ \triangleright \{n_1, \dots, n_k\} \Rightarrow \mathbb{P}[\![S_i]\!]\,(e_{n_1} \sqcap \cdots \sqcap e_{n_k}) \\ \triangleright \{\} \Rightarrow 1 \end{bmatrix}$$
$\quad \triangleright (e_1 \sqcup \cdots \sqcup e_\ell) \Rightarrow \sum_{J \subset [\ell]} \left[(-1)^{|J|-1}\, \mathbb{P}[\![S_1 \otimes \cdots \otimes S_m]\!]\,(\sqcap_{i \in J}\, e_i)\right]$

(f) Sum-Product Expressions (Distribution Semantics)

Listing 1. Syntax and semantics of a core calculus for sum-product expressions and related domains.

Events Lst. 1c shows the Event domain, which specifies predicates on variables. The valuation $\mathbb{E}[\![e]\!]$: Var $\to$ Outcomes of an Event takes a variable x and returns the set $v \in$ Outcomes of elements that satisfy the predicate along dimension x, leveraging the properties of *preimg*. This domain specifies measurable sets of an n-dimensional distribution on variables $\{x_1, \dots, x_n\}$ as follows: let $\sigma_{\text{gen}}(\{A_1, A_2, \dots\})$ be the sigma-algebra generated by $A_1, A_2, \dots$, and define $\mathcal{B}^n(\text{Outcome}) \coloneqq \sigma_{\text{gen}}(\{\prod_{i=1}^n U_i \mid \forall_{1 \leq i \leq n}.\ U_i \in \mathcal{B}(\text{Outcomes})\})$. In other words, $\mathcal{B}^n$(Outcome) is the n-fold product sigma-algebra generated by open rectangles of Outcomes. Any $e \in$ Event specifies a measurable set U in $\mathcal{B}^n$(Outcome), whose ith coordinate $U_i = \mathbb{E}[\![e]\!]\, x_i$ if $x_i \in \textit{vars}\, e$, and U_i = Outcomes otherwise. Any Transform in e is solved and any Var that does not appear in e is marginalized, as in the next example.

Example 3.2. Let {X, Y, Z} be elements of Var. Then

$$e \coloneqq \texttt{Reciprocal(Id(X)) in ((\#f 1) (2 \#f))}$$

corresponds to the $\mathcal{B}^3$(Outcome)-measurable set

$$\{(\downarrow {}^{\text{Real}}_{\text{Outcome}}\, r) \mid 1/2 \leq r \leq 1\} \times \text{Outcomes} \times \text{Outcomes}.$$

As in Remark 3.1, we may speak about probability distributions on $\mathcal{B}^n$(Outcome) as mappings from Event to [0, 1].

Primitive Distributions Lst. 1e presents the primitive distributions out of which multivariate distributions are constructed. The CDF domain contains cumulative distribution functions F, whose quantile function is denoted F^{-1} and derivative F'. CDF is in 1-1 correspondence with all distributions and random variables on Real [7, Thms 12.4, 14.1]. The Distribution domain specifies continuous real, atomic

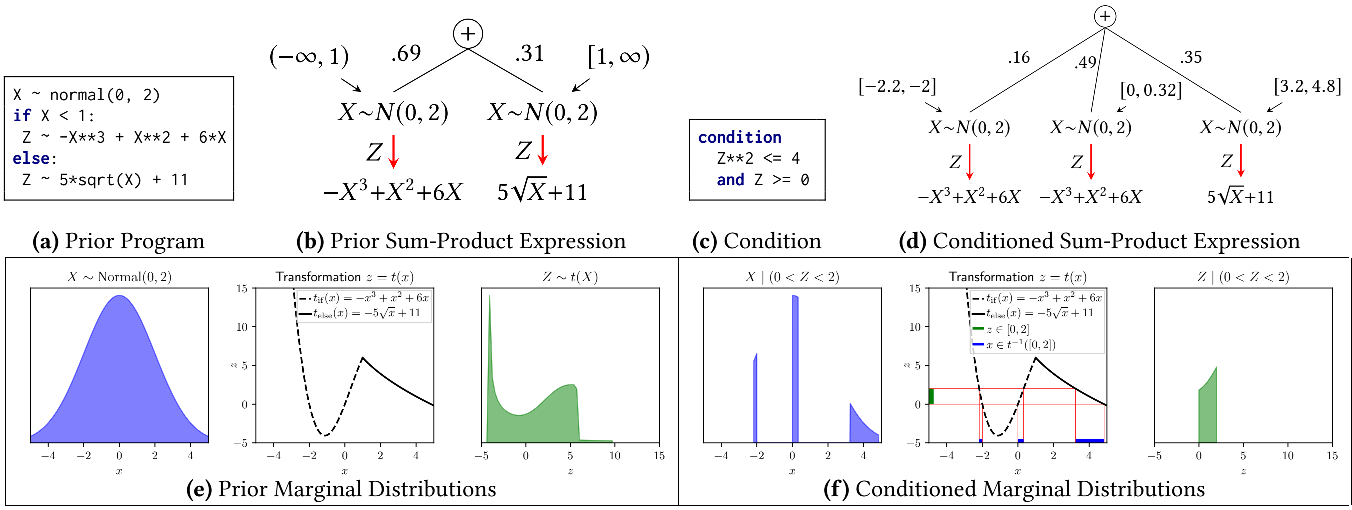


Figure 4. Inference on a stochastic many-to-one transformation of a real random variable in Sppl.

real (on the integers) and nominal distributions. The denotation $\mathbb{D}[\![d]\!]$ of a Distribution is a distribution on Outcomes (recall Remark 3.1). For example, $\texttt{DistR}(F\,r_1\,r_2)$ is the restriction of F to a positive measure interval $[r_1, r_2]$. The distributions specified by DistR and DistI can be simulated using a variant of the integral probability transform (Prop. A.1 in Appx. A), which also defines their sampling semantics.

Sum-Product Expressions Lsts. 1d and 1f show the probability density and distribution semantics of the SPE domain, respectively, whose elements are probability measures. The following conditions specify well-definedness for SPE:

(C1) $\forall\,\texttt{Leaf}(x\ d\ \sigma).\ x \in \sigma$ and $\sigma(x) = \texttt{Id}(x)$.

(C2) $\forall\,\texttt{Leaf}(x\ d\ \sigma)$. If $\mathrm{dom}(\sigma) = \{x, x_1, \dots, x_m\}$ for some $m > 0$ then $\forall_{1 \le t \le m}.\ (\mathit{vars}\ \sigma(x_t)) \subset \{x, x_1, \dots, x_{t-1}\}$.

(C3) $\forall (S_1 \otimes \cdots \otimes S_m).\ \forall i{\neq}j.\ (\mathit{scope}\, S_i) \cap (\mathit{scope}\, S_j) = \varnothing$.

(C4) $\forall (S_1\ w_1) \oplus \cdots \oplus (S_m\ w_m).\ \forall i.\ (\mathit{scope}\, S_i) = (\mathit{scope}\, S_1)$.

(C5) $\forall (S_1\ w_1) \oplus \cdots \oplus (S_m\ w_m).\ w_1 + \cdots + w_n > 0$.

For Leaf, (C1) ensures that σ maps the leaf variable x to the Identity Transform and (C2) ensures there are no cyclic dependencies or undefined variables in Environment σ. Condition (C3) ensures the scopes of all children of a Product are disjoint and (C4) ensures the scopes of all children of a Sum are identical, which together ensure completeness and decomposability from sum-product networks [47, Defs. 4, 5].

In Lst. 1f, the denotation $\mathbb{P}[\![S]\!]$ of $S \in$ SPE is a map from $e \in$ Event to its probability under the n-dimensional distribution defined by S, where $n := |\mathit{scope}\, S|$ is the number of variables in S. A terminal node $\texttt{Leaf}(x\,d\,\sigma)$ is comprised of a Var x, Distribution d, and Environment σ that maps other variables to a Transform of x, e.g., $\texttt{Z} \mapsto \texttt{Poly(Root(Id(X) 2)}\ [11, 5])$.

When assessing the probability of e at a Leaf, *subsenv* (Lst. 13 in Appx. A) rewrites e as an Event e' on one variable x, so that the probability of Outcomes that satisfy e is exactly $\mathbb{D}[\![d]\!]\,(\mathbb{E}[\![e']\!]\,x)$. The *scope* function (Lst. 12 in Appx. A) returns the list of variables in S. For a Sum, the probability of e is a weighted average of the probabilities under each subexpression. For a Product, the semantics are defined in terms of $(\mathit{dnf}\ e)$ (Lst. 15 in Appx. B), leveraging inclusion-exclusion.

In Lst. 1d, the denotation $\mathbb{P}_0[\![S]\!]$ defines the density semantics of SPE, used for measure zero events such as $\{X = 3, Y = \pi, Z = \texttt{"foo"}\}$ under a mixed-type base measure. These semantics, which define the density as a pair, adapt "lexicographic likelihood-weighting", an approximate inference algorithm for discrete-continuous Bayes Nets [68], to exact inference using "lexicographic enumeration" for SPE.

4 Conditioning Sum-Product Expressions

We next present the main theoretical result for exact inference on probability distributions defined by an expression $S \in$ SPE and describe the inference algorithm for conditioning on an Event (Lst. 1c) in the core calculus, which includes transformations and predicates with set-valued constraints.

Theorem 4.1 (Closure under conditioning)**.** *Let $S \in$ SPE and $e \in$ Event be given, where $\mathbb{P}[\![S]\!]\,e > 0$. There exists an algorithm which, given S and e, returns $S' \in$ SPE such that, for all $e' \in$ Event, the probability of e' according to S' is equal to the conditional probability of e' given e according to S, i.e.,*

$$\mathbb{P}[\![S']\!]\,e' \equiv \mathbb{P}[\![S]\!]\,(e' \mid e) := \frac{\mathbb{P}[\![S]\!]\,(e \sqcap e')}{\mathbb{P}[\![S]\!]\,e}. \tag{5}$$

Thm. 4.1 is a structural conjugacy property [20] for the family of probability distributions defined by the SPE domain, where both the prior and posterior are identified by elements of SPE. We establish Eq. (5) constructively, by describing a new algorithm *condition* : SPE $\to$ Event $\to$ SPE that satisfies

$$\mathbb{P}[\![(\mathit{condition}\, S\, e)]\!]\,e' = \mathbb{P}[\![S]\!]\,(e' \mid e) \tag{6}$$

for all $e, e' \in$ Event with $\mathbb{P}[\![S]\!]\,e > 0$. Refer to Appx. D for the proof. Fig. 5 shows a conceptual example of how *condition*

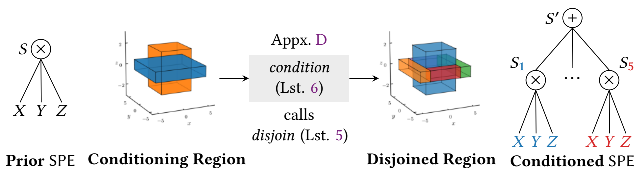


Figure 5. Conditioning a Product S on an Event e that defines a union of hyperrectangles in Real^3. The inference algorithm partitions the region into a disjoint union, in this case converting two overlapping regions into five disjoint regions. The result is a Sum-of-Product, where each child is the restriction of S to one of the disjoint hyperrectangles.

works, where the prior distribution is a Product S and the conditioned distribution is a Sum-of-Product S'. Fig. 4 shows an example of the closure property when the expression has transformed variables (details in Appx. C).

Remark 4.2. Thm. 4.1 refers to a positive probability Event e. As with sum-product networks, SPE is also closed under conditioning on a Conjunction of possibly measure zero equality constraints on non-transformed variables, which appear in many PPL interfaces [17, 41, 51]. Appx. D.3 presents the $condition_0$ algorithm for inference on such events, leveraging the generalized mixed-type density semantics in Lst. 1d.

The next result, Thm. 4.3, states a sufficient requirement for inference using $(condition\, S\, e)$ to scale linearly in the size of S, which holds for both zero and positive measure events.

Theorem 4.3. *The runtime of* $(condition\, S\, e)$ *scales linearly in the number of nodes in the graph representing* S *whenever* e *is a single* Conjunction $(t_1 \,\mathtt{in}\, v_1) \sqcap \cdots \sqcap (t_m \,\mathtt{in}\, v_m)$ *of* Containment *constraints on non-transformed variables.*

5 Translating Probabilistic Programs to Sum-Product Expressions

We next present a probabilistic language called Sppl and show how to translate each program in the language to an element $S \in \text{SPE}$ that symbolically represents (via $\mathbb{P}[\![S]\!]$) the probability distribution specified by the program. As in Fig. 1, S can then be used to answer queries about an Event e:

simulate: Samples from the distribution defined by $\mathbb{P}[\![S]\!]$;
prob: Computes the probability of e, using $\mathbb{P}[\![S]\!]\, e$ (Lst. 1f);
condition: Conditions on e, using *condition* (Eq. (6)).

Lst. 2 shows the source syntax of Sppl, which contains standard constructs of an imperative language such as **array** data structures, **if**-**else** statements, and bounded **for** loops. The **switch**-**case** macro is defined in Eq. (4). Random variables are defined using "sample" (~) and **condition**(E) can be used to restrict executions to those for which $E \in \text{Expr}$ evaluates to #t as part of the prior definition. Lst. 3 defines a relation $\langle C, S\rangle \to_{\text{SPE}} S'$, which translates a "current" $S \in \text{SPE}$ and $C \in \text{Command}$ into $S' \in \text{SPE}$, where the initial

$$
\begin{aligned}
&x \in \text{Var};\ y \in \text{ArrayVar};\ n \in \text{Natural};\ b \in \text{Boolean};\ r \in \text{Real};\ s \in \text{String};\\
&o_{\text{arith}} \in \{+, -, *, /, **\};\quad o_{\text{bool}} \in \{\mathtt{and}, \mathtt{or}\};\quad o_{\text{neg}} \in \{\mathtt{not}\};\\
&o_{\text{rel}} \in \{<=, <, >, >=, ==, \mathtt{in}\};\quad D \in \{\mathtt{normal}, \mathtt{poisson}, \mathtt{choice}, \dots\};\\
&E \in \text{Expr} ::= x \mid n \mid b \mid r \mid s \mid y[E] \mid D(E^*) \mid (E_1, \dots, E_m)\\
&\quad \mid E_1\, o_{\text{arith}}\, E_2 \mid o_{\text{neg}}\, E \mid E_1\, o_{\text{bool}}\, E_2 \mid E_1\, o_{\text{rel}}\, E_2\\
&C \in \text{Command} ::= x = E \mid y[E_1] = E_2 \mid x \sim E \mid y[E_1] \sim E_2 \mid y = \mathtt{array}(E)\\
&\quad \mid \mathtt{skip} \mid C_1 ; C_2 \mid \mathtt{if}\ E\ \{C_1\}\ \mathtt{else}\ \{C_2\} \mid \mathtt{condition}(E)\\
&\quad \mid \mathtt{for}\ x\ \mathtt{in}\ \mathtt{range}(E_1, E_2)\ \{C\} \mid \mathtt{switch}\ x_1\ \mathtt{cases}\ (x_2\ \mathtt{in}\ E)\ \{C\}
\end{aligned}
$$

Listing 2. Source syntax of Sppl.

(Sample)
$$\frac{E \Downarrow d; \qquad \text{where } x \notin \mathit{scope}\, S}{\langle x \sim E, S\rangle \to_{\text{SPE}} S \otimes (x\ d\ \{x \mapsto \mathtt{Id}(x)\})}$$

(Transform-Leaf)
$$\frac{E \Downarrow t; \quad \text{where } \mathit{vars}\, t \in \text{dom}(\sigma),\ x \notin \text{dom}(\sigma)}{\langle x = E, \mathtt{Leaf}(x'\ d\ \sigma)\rangle \to_{\text{SPE}} \mathtt{Leaf}(x'\ d\ (\sigma \cup \{x \mapsto t\}))}$$

(Transform-Sum)
$$\frac{E \Downarrow t,\ \forall_{1\le i\le m}.\langle x = E, S_i\rangle \to_{\text{SPE}} S'_i}{\langle x = E, \oplus_{i=1}^{m}(S_i\ w_i)\rangle \to_{\text{SPE}} \oplus_{i=1}^{m}(S'_i\ w_i)}$$

(Transform-Prod)
$$\frac{E \Downarrow t,\ \langle x = E, S_j\rangle \to_{\text{SPE}} S'_j; \quad \text{where } j := \min\{i \mid (\mathit{vars}\, E) \in \mathit{scope}\ S_i\} > 0}{\langle x = E, \otimes_{i=1}^{m} S_i\rangle \to_{\text{SPE}} \otimes_{i=1, i\ne j}^{m} S_i \otimes S'_j}$$

(IfElse)
$$\frac{E \Downarrow e,\ \langle C_1, \mathit{condition}\, S\, e\rangle \to_{\text{SPE}} S_1,\ \langle C_2, \mathit{condition}\, S\, (\mathit{negate}\, e)\rangle \to_{\text{SPE}} S_2}{\langle \mathtt{if}\ E\ \{C_1\}\ \mathtt{else}\ \{C_2\}, S\rangle \to_{\text{SPE}} (S_1\, \mathbb{P}[\![S]\!]\, e) \oplus (S_2\, (1 - \mathbb{P}[\![S]\!]\, e))}$$

(For-Repeat)
$$\frac{E_1 \Downarrow n_1,\ E_2 \Downarrow n_2; \qquad \text{where } n_1 < n_2}{\begin{aligned}&\langle \mathtt{for}\ x\ \mathtt{in}\ \mathtt{range}(E_1, E_2)\ \{C\}, S\rangle\\ &\quad \to_{\text{SPE}} \langle C[x/n_1];\ \mathtt{for}\ x\ \mathtt{in}\ \mathtt{range}(n_1 + 1, E_2)\ \{C\}, S\rangle\end{aligned}}$$

Listing 3. Example rules for translating an Sppl command C (Lst. 2) to an element of SPE (Lst. 1f).

step operates on an "empty" S. (Lst. 8 in Appx. E defines a semantics-preserving inverse of $\to_{\text{SPE}}$). The $\Downarrow$ relation evaluates $E \in \text{Expr}$ to other domains in the core calculus using straightforward rules. We briefly describe key rules of $\to_{\text{SPE}}$:

(Transform-Leaf) updates the environment σ at each Leaf.
(Transform-Sum) delegates to all subexpressions.
(Transform-Prod) delegates to the subexpression whose scope contains the transformed variable.
(For-Repeat) unrolls a **for** loop into a Command sequence.
(IfElse) returns a Sum with two subexpressions, where the **if** branch is conditioned on the test Event and the **else** branch is conditioned on the negation of the test Event. This translation step involves running inference (using *condition*, Eq. (6)) on the current $S \in \text{SPE}$ translated so far.

The rule for **condition**(E) (not shown) calls $(condition\, S\, e)$ (Eq. 6) or $(condition_0\, S\, e)$ (Remark 4.2), where $E \Downarrow e$. To ensure Sppl programs translate to well-defined element of SPE, per (C1)–(C5)), each program must satisfy these restrictions:

(R1) Variables x in $x \sim E$ (Sample) and $x = E$ (Transform-Leaf) must be fresh (ensures conditions (C1), (C2) and (C3)).
(R2) The branches in an **if**-**else** statement must define identical variables (ensures conditions (C4) and (C5)).

(a) Invalid program (translates to an infinite-sized SPE)

```
mu ~ beta(a=4, b=3, scale=7)
num_loops ~ poisson(mu)       # invalid (real integral)
for i in range(0, num_loops): # invalid (infinite series)
  [... commands ... ]
```

(b) Valid program (translates to a finite-sized SPE)

```
mu ~ beta(a=4, b=3, scale=7)
# binspace partitions [0,7] into 10 intervals
switch (mu) cases (m in binspace(0, 7, n=10)):
  num_loops ~ poisson(m.mean()) # discretization
condition (num_loops < 50)      # truncation
switch num_loops cases (n in range(50)):
  for i in range(0, n):
    [... commands ... ]
```

Listing 4. Examples of valid and invalid Sppl programs.

(R3) Derived random variables are obtained via (many-to-one) *univariate* transformations (Lst. 1b).
(R4) Parameters of distributions D or **range** must be either constants or random variables with finite support.

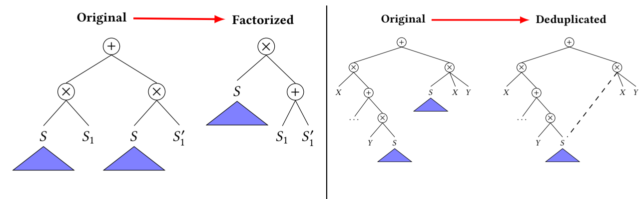
(R3) is required since the distribution of a multivariate transform (e.g., $Z = X/Y^2$) is typically intractable and does not factor into Sum and Product expressions. (R4) is required to ensure a finite-size SPE: distributional parameters with infinite support require integrals (uncountable support) or infinite series (countable support), which are not in SPE. Lst. 4 shows an example of using **switch** and **condition** to work around these restrictions by discretization and truncation.

5.1 Building Compact Sum-Product Expressions

As discrete Bayesian networks can be encoded as Sppl programs, it is possible to write programs where exact inference is NP-Hard [16], which corresponds to an element of SPE that is exponentially large. It is well known that the complexity of exact inference in Bayesian networks is worst case exponential in the treewidth, which is the only structural restriction that can ensure tractability [12]. As computing treewidth is NP-Complete [3], for fundamental theoretical reasons we cannot generally check conditions needed for even simple Sppl programs, such as those that only use if/else statements on binary variables, to translate into a "small" expression.

However, many models of interest contain (conditional) independence relationship [33] that induce a compact factorization of the model into tractable subparts, as in, e.g., Sec. 2.2. Sppl uses several optimization techniques to improve scalability of translation (Lst. 3) and inference (Eq. (6)) by automatically exploiting independences and repeated structure, when they exist, to build compact sum-product expressions.

Factorization Using standard algebraic manipulations, a sum-product expression can be made smaller without changing its semantics (Lst. 1f) by "factoring out" common terms (Fig. 6a), provided that the new expression satisfies (C1)–(C5). Factorization plays a key role in the (IfElse) rule of $\to_{\text{SPE}}$: since all statements before the **if**-**else** are shared by the bodies of the **if** and **else** branches, statements outside the branch that are independent of statements inside the branch often produce subexpressions that can be factored out.



(a) Factorization **(b)** Deduplication

Figure 6. Exploiting independences and repeated structure during translation of Sppl programs to build compact sum-product expressions. Blue subtrees are identical components.

Table 1. Measurements of SPE graph size with and without the factorization and deduplication optimizations in Fig. 6.

Benchmark	No. of Nodes in Translated SPE		Data Compression Ratio (unopt/opt)
	Unoptimized	Optimized	
Hiring [2]	33	27	1.2x
Alarm [46]	58	45	1.3x
Grass [46]	130	59	2.2x
Noisy OR [46]	783	132	4.1x
Clinical Trial [46]	43761	4131	10.6x
Heart Disease [56]	1041235	6257	166.4x
Hierarchical HMM (Sec. 2.2)	29273397577908185	1787	16381308101795x

Deduplication When a sum-product expression contains duplicate subexpressions that cannot be factored out without violating the definedness conditions, we instead resolve duplicates into a single physical representative. Fig. 6b shows an example where the left and right components of the original expression contain an identical subexpression S (in blue), but factorization would lead to an invalid sum-product expression. The optimizer represents the computation graph of this expression using a single data structure S shared by the left and right subtrees (see also Figs. 3c–3d).

Memoization While deduplication reduces memory overhead, memoization is used to reduce runtime overhead. Consider either SPE in Fig. 6b: calling *condition* on the Sum root will dispatch the query to the left and right subexpressions (Lst. 6b). We cache the results of (*condition* S e) or $\mathbb{P}\,[\![S]\!]\,e$ when S is visited in the left subtree to avoid recomputing the result when S is visited again in the right subtree via a depth-first traversal. Memoization delivers large runtime gains not only for solving queries but also for detecting duplicates returned by *condition* in the (IfElse) translation step.

Measurements Table 1 shows measurements of performance gains delivered by the factorization and deduplication optimizations on seven benchmarks. Compression ratios range between 1.2x to 1.64×10^{13}x and are highest in the presence of independence or repeated structure. The deduplication and memoization optimizations together enable fast detection of duplicate subtrees by comparing logical memory addresses of internal nodes in $O(1)$ time, instead of computing hash functions that require an expensive subtree traversal.

Table 2. Runtime measurements and speedup for 15 fairness verification tasks using Sppl, FairSquare [2], and VeriFair [4].

Decision Program	Population Model	Lines of Code	Fairness Judgment	Wall-Clock Runtime (seconds) FairSquare	VeriFair	Sppl	Sppl Speedup Factor vs. FairSquare	vs. VeriFair
DT_4	Independent	15	Unfair	1.4	16.0	0.01	**140x**	**1600x**
	Bayes Net. 1	25	Unfair	2.5	1.27	0.03	**83x**	**42x**
	Bayes Net. 2	29	Unfair	6.2	0.91	0.03	**206x**	**30x**
DT_{14}	Independent	32	Fair	2.7	105	0.03	**90x**	**3500x**
	Bayes Net. 1	46	Fair	15.5	152	0.07	**221x**	**2171x**
	Bayes Net. 2	50	Fair	70.1	151	0.08	**876x**	**1887x**
DT_{16}	Independent	36	Fair	4.1	13.6	0.03	**136x**	**453x**
	Bayes Net. 1	49	Unfair	12.3	1.58	0.08	**153x**	**19x**
	Bayes Net. 2	53	Unfair	30.3	2.02	0.08	**378x**	**25x**
DT_{16}^{α}	Independent	62	Fair	5.1	2.01	0.06	**85x**	**33x**
	Bayes Net. 1	58	Fair	15.4	21.6	0.12	**128x**	**180x**
	Bayes Net. 2	45	Fair	53.8	24.5	0.12	**448x**	**204x**
DT_{44}	Independent	93	Fair	15.6	23.1	0.05	**312x**	**462x**
	Bayes Net. 1	109	Unfair	264.1	19.8	0.09	**2934x**	**220x**
	Bayes Net. 2	113	Unfair	t/o	20.1	0.09	—	**223x**

6 Evaluation

We implemented a prototype of Sppl[2] and evaluated its performance on benchmark problems from the literature. Sec. 6.1 compares the runtime of verifying fairness properties of decision trees using Sppl to FairSquare [2] and VeriFair [4], two state-of-the-art fairness verification tools. Sec. 6.2 compares the runtime of conditioning and querying probabilistic programs using Sppl to PSI [23], a state-of-the-art tool for exact symbolic inference. Sec. 6.3 compares the runtime of computing exact rare event probabilities in Sppl to sampling-based estimation in BLOG [40]. Experiments were run on Intel i7-8665U 1.9GHz CPU with 16GB RAM.

6.1 Fairness Benchmarks

Characterizing the fairness of classification algorithms is a growing application area in machine learning [21]. Recently, Albarghouthi et al. [2] precisely cast the problem of verifying the fairness of a classifier in terms of computing ratios of conditional probabilities in a probabilistic program that specifies the data generating and classification processes. Briefly, if (i) D is a decision program that classifies whether applicant A should be hired; (ii) H is a population program that generates random applicants; and (iii) ϕ_{m} (resp. ϕ_{q}) is a predicate on A that is true if the applicant is a minority (resp. qualified), then D is ϵ-fair on H (where $\epsilon > 0$) if

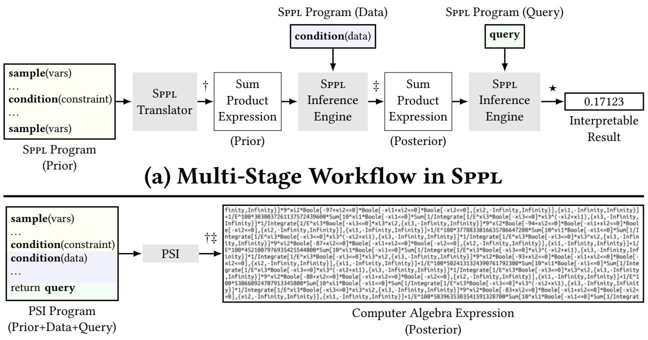
$$\frac{\Pr_{A\sim H}\left[D(A) \mid \phi_{\mathrm{m}}(A) \wedge \phi_{\mathrm{q}}(A)\right]}{\Pr_{A\sim H}\left[D(A) \mid \neg\phi_{\mathrm{m}}(A) \wedge \phi_{\mathrm{q}}(A)\right]} > 1 - \epsilon, \tag{7}$$

i.e., the probability of hiring a qualified minority applicant is ϵ-close to that of hiring a qualified non-minority applicant.

In this evaluation, we compare the runtime needed by Sppl to obtain a fairness judgment (Eq. (7)) for machine-learned decision and population programs against the FairSquare [2] and VeriFair [4] solvers. We evaluate performance on the decision tree benchmarks from Albarghouthi et al. [2, Sec. 6.1], which are one-third of the full benchmark set. Sppl cannot solve the neural network and support-vector machine benchmarks, as they contain multivariate transforms which do not have exact tractable solutions and are ruled out by the Sppl restriction (R3). FairSquare and VeriFair can express these benchmarks as they have approximate inference.

Table 2 shows the results. The first column shows the decision making program (DT_n means "decision tree" with n conditionals); the second column shows the population model used to generate data; the third column shows the lines of code (in Sppl); and the fourth column shows the result of the fairness analysis (FairSquare, VeriFair, and Sppl produce the same judgment on all fifteen benchmarks). The remaining columns show the runtime and speedup factors. We note that Sppl, VeriFair, and FairSquare are all implemented in Python, which allows for a fair comparison. The measurements indicate that Sppl consistently obtains probability estimates in milliseconds, whereas the two baselines can each require over 100 seconds. The Sppl speedup factors are up to 3500x (vs. VeriFair) and 2934x (vs. FairSquare). We further observe that the runtimes in FairSquare and VeriFair vary significantly. For example, VeriFair uses rejection sampling to estimate Eq. (7) with a stopping rule to determine when the estimate is close enough, leading to unpredictable runtime (e.g., >100 seconds for DT_{14} but <1 second for DT_4, Bayes Net. 2). FairSquare, which uses symbolic volume computation and hyperrectangle sampling to approximate Eq. (7), is faster than VeriFair in some cases (e.g., DT_{14}), but times out in others (DT_{44}, Bayes Net. 2). In contrast, Sppl, computes exact probabilities for Eq. (7) and its runtime does not vary significantly across the various benchmark problems. The performance–expressiveness trade-off here is that Sppl computes exact probabilities and is substantially faster on the decision tree problems that it can express. FairSquare and VeriFair compute approximate probabilities that enable them to express more fairness problems, at the cost by of a higher and less predictable runtime on the decision trees.

[2]Available in supplement and online at https://github.com/probcomp/sppl.



(a) Multi-Stage Workflow in Sppl

(b) Single-Stage Workflow in PSI

Figure 7. Comparison of multi-stage and single-stage inference workflows. In Sppl, modeling, observing data, and querying are separated into distinct stages, enabling substantial efficiency gains from computation reuse across multiple datasets and/or queries, as opposed to single-stage workflows such as in PSI which combine all these tasks into one large symbolic computation (daggers/colors used in Table 4).

6.2 Comparison to Symbolic Integration

We next compare Sppl to PSI [23], a state-of-the-art symbolic inference engine, on benchmark problems that include discrete, continuous, and transformed random variables. PSI can express more inference problems than Sppl, as it uses general computer algebra without having restrictions (R3) and (R4) in Sppl. As a result, Sppl can solve 14/21 benchmarks listed in [23, Table 1]. We first discuss key architecture novelties in Sppl that contribute to its performance gains.

Workflow Comparison In Sppl, the multi-stage modeling and inference workflow (Fig. 7a) involves three steps that reflect the key elements of a Bayesian inference problem:

(S1) Translating the model program into a prior SP S.
(S2) Conditioning S on data to obtain a posterior SP S'.
(S3) Querying S', using, e.g., `prob` or `simulate`.

An advantage of this multi-stage workflow is that multiple tasks can be run at a given stage without rerunning previous stages. For example, multiple datasets can be observed in (S2) without translating the prior expression in (S1) once per dataset; and, similarly, multiple queries can be run in (S3) without conditioning on data in (S2) once per query. In contrast, PSI adopts a single-stage workflow (Fig. 7b), where a single program contains the prior distribution over variables, "observe" (i.e., "condition") statements for conditioning on a dataset, and a "return" statement for the query. PSI converts the program into a symbolic expression for the distribution over the return value: if this expression is "complete" (i.e., no unevaluated symbolic integrals) it can be used to obtain interpretable answers (e.g., for plotting or tabulating); otherwise, the result is "partial" and is too complex to be used for practical purposes. A consequence of the single-stage workflow in a system like PSI is that the entire solution is recomputed from scratch on a per-dataset or per-query basis.

Table 3. Distribution of end-to-end inference runtime for four benchmarks from Table 4 using PSI [23] and Sppl.

Benchmark	**Mean/Std Runtime (sec/sec)**	
	PSI	Sppl
Digit Recognition	26.5/1.3	15.9/0.5
Markov Switching	22.5/3.8	0.1/0.0
Student Interviews	539/663	7.8/0.2
Clinical Trial	107.3/153.2	12.7/0.3

Runtime Comparison Table 4 compares the runtime of Sppl and PSI on seven benchmarks problems: Digit Recognition [23]; TrueSkill [36]; Clinical Trial [23]; Gamma transforms (described below); Student Interviews [36] (two variants); and Markov Switching (two variants, from Sec. 2.2); The second column shows the distributions in each benchmark, which include continuous, discrete, and transformed variables. The third column shows the number of datasets on which to condition the program. The next three columns show the time needed to translate the program (stage (S1)), condition the program on a dataset (stage (S2)), and query the posterior (stage (S3))—entries in the latter two columns are written as $n \times t$, where n is the number of datasets and t the average time per dataset. For PSI: (i) modeling and observing data are a single stage, shown in the merged gray cell; and (ii) querying the posterior times out whenever the system returns a result with unsimplified integrals (⋉). The last column shows the overall runtime for solving all n tasks.

For benchmarks that both systems solve completely, Sppl realizes speedups between 3x (Digit Recognition) to 3600x (Markov Switching$_3$). In addition, the measurements show the advantage of our multi-stage workflow; for example, in TrueSkill, which uses a Poisson–Binomial distribution, Sppl translation (3.4 seconds) is more expensive than both conditioning on data (0.7 seconds) and querying (0.1 seconds), which highlights the benefit of amortizing the translation cost over several datasets or queries. In PSI, solving TrueSkill takes 2×41.6 seconds, but the solution contains unsimplified integrals and is thus unusable. The Markov Switching and Student Interviews benchmarks show that PSI may not perform well in the presence of many discrete random variables.

The Gamma Transform benchmark tests the robustness of many-to-one transforms of random variables (Lst. 1b), where $X \sim \text{Gamma}(3, 1)$; $Y = 1/\exp X^2$ if $X < 1$ and $Y = 1/\ln X$ otherwise; and $Z = -Y^3 + Y^2 + 6Y$. Each of the $n = 5$ datasets specifies a different constraint $\phi(Z)$ and a query about the posterior $Y \mid \phi(Z)$, which needs to compute and integrate out $X \mid \phi(Z)$. PSI reports that there is an error in its answer for all five datasets, whereas Sppl, using the symbolic transform solver from Appx. C.2, solves all five problems effectively.

Table 3 compares the runtime variance using Sppl and PSI for four of the benchmarks in Table 4, repeating one query over 10 datasets. In all benchmarks, the Sppl variance is lower than that of PSI, with a maximum standard deviation $\sigma = 0.5$ sec. In contrast, the spread of PSI runtime is high

Table 4. Runtime comparison of PSI [23] and Sppl.

Benchmark	Distribution	Datasets ◇	System	Wall-Clock Runtime of Inference Stages: Translation †	Conditioning ‡	Querying ★	Overall Time
Digit Recognition	$C{\times}B^{784}$	10	Sppl	6.9 sec	10×7.7 sec	$10\times(<0.01$ sec)	84 sec
			PSI	10×24.3 sec		$10\times(<0.01$ sec)	244 sec
TrueSkill	$P{\times}Bi^{2}$	2	Sppl	3.4 sec	2×0.7 sec	2×0.1 sec	4.9 sec
			PSI	2×41.60 sec		⋉	⊘
Clinical Trial	$B{\times}U^{3}$ $\times B^{50}{\times}B^{50}$	10	Sppl	9.5 sec	10×2.2 sec	$10\times(<0.01$ sec)	31 sec
			PSI	10×107.3 sec		$10\times(<0.01$ sec)	1073 sec
Gamma Transforms	$G{\times}T$ $\times(T{+}T)$	5	Sppl	0.02 sec	5×0.52 sec	5×0.03 sec	2.8 sec
			PSI	5×0.68 sec; i/e		⋉	⊘
Student Interviews$_2$	$P{\times}B^{2}{\times}Bi^{4}$ $\times(A{+}Be)^{2}$	10	Sppl	4.0 sec	10×0.7 sec	10×0.2 sec	13.5 sec
			PSI	10×540 sec; h/m (35GB)		⋉	⊘
Student Interviews$_{10}$	$P{\times}B^{10}{\times}Bi^{20}$ $\times(A{+}Be)^{10}$	10	Sppl	24.6 sec	10×3.9 sec	10×1.2 sec	75 sec
			PSI	o/m (64GB+)		⊘	⊘
Markov Switching$_3$	$B{\times}B^{3}$ $\times N^{3}{\times}P^{3}$	10	Sppl	0.05 sec	$10\times(<0.01$ sec)	$10\times(<0.01$ sec)	0.5 sec
			PSI	10×182.9 sec		$10\times(<0.01$ sec)	1829 sec
Markov Switching$_{100}$	$B{\times}B^{100}$ $\times N^{100}{\times}P^{100}$	10	Sppl	4.1 sec	10×6.5 sec	10×0.5 sec	74 sec
			PSI	o/m (64GB+)		⊘	⊘

Legend

A: Atomic B: Bernoulli Be: Beta
Bi: Binomial C: Categorical
N: Normal G: Gamma P: Poisson
T: Transform U: Uniform

◇ Number of distinct datasets on which to condition the benchmark program.

†, ‡ Runtime of first two phases in Fig. 7; PSI implements these phases in a single computation.

★ Runtime of final phase in Fig. 7; same query used for all datasets of a given benchmark program.

h/m High-Memory
o/m Out-of-Memory
i/e Integration Error
⋉ Unsimplified Symbolic Integrals
⊘ No Value

for Student Interviews (σ = 540 sec, range 64–1890 sec) and Clinical Trial (σ = 153 sec, range 2.75–470 sec). In PSI, the symbolic analyses are sensitive to the numeric values in the dataset, leading to unpredictable runtime across different datasets, even for a fixed query pattern. In Sppl, the runtime depends only on the query pattern not the observed data and therefore behaves predictably across different datasets.

As with the fairness benchmarks in Sec. 6.1, PSI trades off expressiveness with efficacy on tractable problems, and our measurements show that its runtime and memory do not scale well or are unpredictable on benchmarks that Sppl solves very efficiently. Moreover, the evaluations show that PSI can return unusable inference results to the user and that it needs to recompute entire symbolic solutions from scratch for each new dataset or query, whereas Sppl is less expressive than PSI but carries neither of these limitations.

6.3 Comparison to Sampling-Based Estimates

We next compare the runtime and accuracy of estimating probabilities of rare events in a canonical Bayesian network [33] using Sppl and BLOG [40]. As discussed by Koller and Friedman [33, Sec 12.13], rare events are the rule, not the exception, in many applications, as the probability of a predicate $\phi(X)$ decreases exponentially with the number of observed variables in X. Small estimation errors can magnify substantially when, e.g., taking ratios of probabilities.

In Fig. 8, each subplot shows the runtime and probability estimates for a low-probability predicate ϕ. In BLOG, the rejection sampler estimates the probability of ϕ by computing the fraction of times it holds in a size n i.i.d. random sample from the prior. The horizontal red line shows the "ground truth" probability. The x marker shows the runtime needed by Sppl to (exactly) compute the probability and the dots show the estimates from BLOG with increasing runtime (i.e., more samples n). Sppl consistently returns an exact answer in less than 2ms. The accuracy of BLOG estimates improve as the runtime increases: by the strong law of large numbers, these estimates converge to the true value, but the fluctuations for any single run can be large (the standard error decays as $1/\sqrt{n}$). Each "jump" corresponds to a new sample $X^{(j)}$ that satisfies $\phi(X^{(j)})$, which increases the estimate. Without ground truth, it is hard to predict how much computation is needed for BLOG to obtain accurate results: estimates for predicates with $\log p = -12.73$ and $\log p = -17.32$ did not converge within the allotted time, while those for $\log p = -14.48$ converged after 180 seconds.

7 Related Work

Sppl is distinguished by being the first system to deliver exact symbolic inference by translating probabilistic programs to sum-product expressions, which extend and generalize sum-product networks. We briefly discuss related approaches.

Symbolic Integration Several systems deliver exact inference by translating a probabilistic program and observed dataset into a symbolic expression whose solution is the answer to the query [6, 10, 23, 43, 69]. Our approach to exact inference, which uses sum-product expressions instead of general computer algebra, enables effective performance on a range of models and queries, primarily at the expense of the expressiveness of the language on continuous priors. The state-of-the-art solver, PSI [23], can effectively solve many inference problems that Sppl cannot express due to restrictions (R1)–(R4), including higher-order programs [24]. However, comparisons on benchmarks that Sppl targets (Sec. 6.2) find PSI has less scalable and higher variance runtime, and can return partial results with unsimplified symbolic integrals. In contrast, Sppl exploits conditional independences, when they exist, to improve scalability (Sec. 5.1) and delivers complete, usable answers to users. Moreover, Sppl's multi-stage workflow (Fig. 7) allows expensive computations such as translation and conditioning to be amortized over multiple datasets or queries, whereas PSI recomputes the symbolic solution from scratch each time. Hakaru [43] is a symbolic solver that delivers exact inference in a multi-stage workflow based on program transformations, and can disintegrate against a variety of base measures [44]. This paper compares

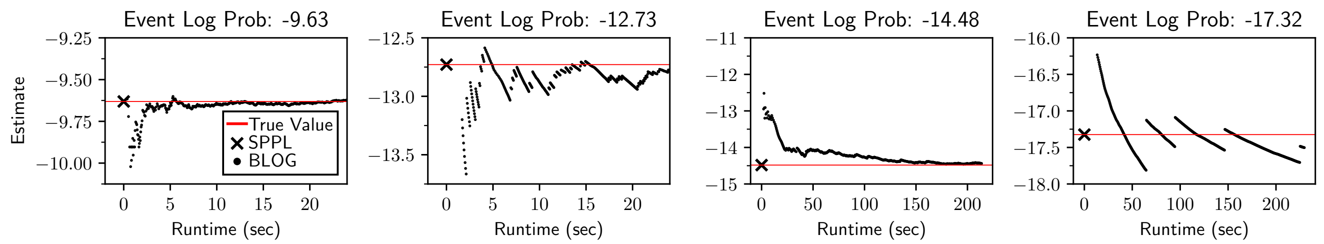


Figure 8. Runtime comparison for computing probabilities using exact inference in Sppl and rejection sampling in BLOG.

against PSI because the reference Hakaru implementation crashes or delivers incorrect or partial results on several benchmark problems [23, Table 1], and, as mentioned by the developers, does not support constructs such as arrays needed to support dozens or hundreds of observations.

Symbolic Execution and Volume Computation: Previous work has addressed the problem of computing the probability of a predicate by integrating a distribution defined by a program [2, 25, 55, 61]. For example, Geldenhuys et al. [25] present a probabilistic symbolic execution technique that uses model counting to compute path probabilities, assuming that all program variables are discrete and uniformly distributed. While Sppl can model a variety of distributions, due to restriction (R3) it only supports predicates that specify rectangular regions, whereas several of the aforementioned systems can (approximately) handle non-rectangular regions. More specifically, predicates in Sppl may include combinations of nonlinear transforms, each of a single variable, which are solved into linear expressions that specify unions of disjoint hyperrectangles (Appx. C.2). Table 2 shows that Sppl delivers substantial speedup on the hyperrectangular regions specified by the important class of decision trees, which are widely used in interpretable machine learning applications.

Sum-Product Networks: The SPFlow library [41] is an object-oriented "graphical model toolkit" in Python for constructing and querying sum-product networks. Sppl leverages a new and more general sum-product representation (Lst. 1) and solves probability and conditioning queries that are not supported by SPFlow (Thm. 4.1), which include mixed random variables, numeric transforms, and logical predicates with set-valued constraints. In addition, we introduce a novel translation strategy (Sec. 5) that allows users to specify models as generative code in a PPL (using e.g., variables, arrays, arithmetic and logical expressions, loops, branches) without having to manually manipulate low-level data structures. "Factored sum-product networks" [58] have been used as intermediate representations for converting a probabilistic program and any functional interpreter into a system of equations whose solution is the marginal probability of the program's return value. These algorithms handle recursive procedures and leverage dynamic programming, but only apply to discrete variables, cannot handle transforms, and require solving fixed-points. Moreover, they have not been quantitatively evaluated on PPL benchmark problems.

Weighted Model Counting/Integration: A common approach to probabilistic inference is using algorithmic reductions from probabilistic programs to weighted-model counting (WMC) or integration (WMI) via knowledge compilation [5, 15, 19, 22, 66]. For example, Symbo [70] leverages WMI for exact inference in hybrid domains, using sentinel decision diagrams as the representation and the PSI solver to symbolically integrate over continuous variables. Dice [30] leverages WMC for scaling exact inference in discrete probabilistic programs and uses binary decision diagram representations that automatically exploit program structure to factorize inference. The representations in Dice enable substantial computation reuse for querying and/or conditioning, such as computing "all-marginal" probabilities by reusing the same compiled representation multiple times. Sppl also leverages factorization and computation reuse, but uses a different representation based on sum-product expressions that handle additional computations such as numeric transforms and continuous and mixed-type random variables.

Probabilistic Program Synthesis: Existing PPL synthesis systems for tabular data [13, 53] produce programs in languages that are subsets of Sppl, which enable automatic synthesis of full Sppl programs from data. Sppl can also unify and extend custom PPL query engines used in these systems for tasks such as similarity search and dependence detection [49, 50, 52]. It may also be fruitful to use structure discovery methods for time series [1, 54] or relational data [32] to synthesize Sppl programs for these domains.

8 Conclusion

We have presented Sppl, a new system that automatically delivers exact answers to a range of probabilistic inference queries. A key insight in Sppl is to impose restrictions on probabilistic programs that enable them to be translated to sum-product expressions, which are highly effective representations for inference. Our evaluation highlights the efficacy of Sppl on inference tasks in the literature and underscores the importance of key design decisions, including the multi-stage inference workflow and techniques used to build compact expressions by exploiting probabilistic structure. In addition to its efficacy as a standalone language, we further anticipate that Sppl could be useful as an embedded domain-specific language within more expressive PPLs, combining the benefits of exact and approximate inference.

References

[1] Amr Ahmed and Eric Xing. 2008. Dynamic Non-Parametric Mixture Models and The Recurrent Chinese Restaurant Process: With Applications to Evolutionary Clustering. In *SDM 2008: Proc. 2008 SIAM Int. Conf. Data Min.* SIAM, Philadelphia, PA, USA, 219–230. https://doi.org/10.1137/1.9781611972788.20

[2] Aws Albarghouthi, Loris D'Antoni, Samuel Drews, and Aditya V. Nori. 2017. FairSquare: Probabilistic Verification of Program Fairness. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 80 (Oct. 2017), 30 pages. https://doi.org/10.1145/3133904

[3] Stefan Arnborg, Derek G Corneil, and Andrzej Proskurowski. 1987. Complexity of Finding Embeddings in a k-Tree. *SIAM J. Alg. Disc. Meth.* 8, 2 (April 1987), 277–284. https://doi.org/10.1137/0608024

[4] Osbert Bastani, Xin Zhang, and Armando Solar-Lezama. 2019. Probabilistic Verification of Fairness Properties via Concentration. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 118 (Oct. 2019), 27 pages. https://doi.org/10.1145/3360544

[5] Vaishak Belle, Andrea Passerini, and Guy Van den Broeck. 2015. Probabilistic Inference in Hybrid Domains by Weighted Model Integration. In *IJCAI 2015: Proc. 24th Int. Jt. Conf. Artif. Intell.* International Joint Conferences on Artificial Intelligence, Cham, 2770–2776.

[6] Sooraj Bhat, Johannes Borgström, Andrew D. Gordon, and Claudio Russo. 2013. Deriving Probability Density Functions from Probabilistic Functional Programs. In *TACAS 2013: Proc. 19th Int. Conf. Tools Alg. Constr. Anal. Syst.* Springer, Berlin, 508–522. https://doi.org/10.1007/978-3-642-36742-7_35

[7] Patrick Billingsley. 1986. *Probability and Measure* (2nd ed.). John Wiley & Sons, New York.

[8] Andrew D. Bolton, Martin Haesemeyer, Josua Jordi, Ulrich Schaechtle, Feras A. Saad, Vikash K. Mansinghka, Joshua B. Tenenbaum, and Florian Engert. 2019. Elements of a Stochastic 3D Prediction Engine in Larval Zebrafish Prey Capture. *eLife* 8 (Nov. 2019), e51975. https://doi.org/10.7554/eLife.51975

[9] Stephen P. Brooks and Andrew Gelman. 1998. General Methods for Monitoring Convergence of Iterative Simulations. *J. Comput. Graph. Stat.* 7, 4 (1998), 434–455. https://doi.org/10.1080/10618600.1998.10474787

[10] Jacques Carette and Chung-chieh Shan. 2016. Simplifying Probabilistic Programs Using Computer Algebra. In *PADL 2016: Proc. 18th Int. Sympos. Pract. Asp. Declar. Lang.* Springer, Cham, 135–152. https://doi.org/10.1007/978-3-319-28228-2_9

[11] Bob Carpenter, Andrew Gelman, Matthew Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2017. Stan: A Probabilistic Programming Language. *J. Stat. Soft.* 76, 1 (Jan. 2017), 1–32. https://doi.org/10.18637/jss.v076.i01

[12] Venkat Chandrasekeran, Nathan Srebro, and Prahladh Harsha. 2008. Complexity of Inference in Graphical Models. In *UAI 2008: Proc. 24th Annu. Conf. Uncertain. Artif. Intell.* AUAI Press, Arlington, VA, USA, 70–78.

[13] Sarah Chasins and Phitchaya M. Phothilimthana. 2017. Data-driven Synthesis of Full Probabilistic Programs. In *CAV 2017: Proc. 29th Int. Conf. Comput. Aided Verif.* Springer, Cham, 279–304. https://doi.org/10.1007/978-3-319-63387-9_14

[14] Nick Chater, Joshua B. Tenenbaum, and Alan Yuille. 2006. Probabilistic Models of Cognition: Conceptual Foundations. *Trends Cogn. Sci.* 10, 7 (July 2006), 287–291. https://doi.org/10.1016/j.tics.2006.05.007

[15] Mark Chavira and Adnan Darwiche. 2008. On Probabilistic Inference by Weighted Model Counting. *Artif. Intell.* 172, 6 (April 2008), 772–799. https://doi.org/10.1016/j.artint.2007.11.002

[16] Gregory F. Cooper. 1990. The Computational Complexity of Probabilistic Inference Using Bayesian Belief Networks. *Artif. Intell.* 42, 2 (March 1990), 393–405. https://doi.org/10.1016/0004-3702(90)90060-D

[17] 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 *PLDI 2019: Proc. 40th ACM SIGPLAN Conf. Program. Lang. Des. Impl.* ACM, New York, NY, USA, 221–236. https://doi.org/10.1145/3314221.3314642

[18] Paul Dagum and Michael Luby. 1993. Approximating Probabilistic Inference in Bayesian Belief Networks is NP-Hard. *Artif. Intell.* 60, 1 (March 1993), 141–153. https://doi.org/10.1016/0004-3702(93)90036-B

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

[20] Persi Diaconis and Donald Ylvisaker. 1979. Conjugate Priors for Exponential Families. *Ann. Statist.* 7, 2 (March 1979), 269–281. https://doi.org/10.1214/aos/1176344611

[21] Cynthia Dwork, Moritz Hardt, Toniann Pitassi, Omer Reingold, and Richard Zemel. 2012. Fairness through Awareness. In *ITCS 2012: Proc. 3rd Innov. Theor. Comput. Sci. Conf.* ACM, New York, NY, USA, 214–226. https://doi.org/10.1145/2090236.2090255

[22] Daan Fierens, Guy Van den Broeck, Ingo Thon, Bernd Gutmann, and Luc De Raedt. 2011. Inference in Probabilistic Logic Programs using Weighted CNF's. In *UAI 2011: Proc. 27th Annu. Conf. Uncertain. Artif. Intell.* AUAI Press, Arlington, VA, USA, 211–220.

[23] Timon Gehr, Sasa Misailovic, and Martin Vechev. 2016. PSI: Exact Symbolic Inference for Probabilistic Programs. In *CAV 2016: Proc. 28th Int. Conf. Comput. Aided Verif.* Springer, Cham, 62–83. https://doi.org/10.1007/978-3-319-41528-4_4

[24] Timon Gehr, Samuel Steffen, and Martin Vechev. 2020. λPSI: Exact Inference for Higher-Order Probabilistic Programs. In *PLDI 2020: Proc. 41st ACM SIGPLAN Conf. Program. Lang. Des. Impl.* ACM, New York, NY, USA, 883–897. https://doi.org/10.1145/3385412.3386006

[25] Jaco Geldenhuys, Matthew B. Dwyer, and Willem Visser. 2012. Probabilistic Symbolic Execution. In *ISSTA 2012: Proc. 2012 Int. Sympos. Soft. Test. Anal.* ACM, New York, NY, USA, 166–176. https://doi.org/10.1145/2338965.2336773

[26] Robert Gens and Pedro Domingos. 2013. Learning the Structure of Sum-Product Networks. In *ICML 2013: Proc. 30th Int. Conf. Mach. Learn.* PMLR, 873–880.

[27] Zoubin Ghahramani. 2015. Probabilistic Machine Learning and Artificial Intelligence. *Nature* 521, 7553 (May 2015), 452–459. https://doi.org/10.1038/nature14541

[28] Noah Goodman, Vikash Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. 2008. Church: A Language For Generative Models. In *UAI 2008: Proc. 24th Annu. Conf. Uncertain. Artif. Intell.* AUAI Press, Arlington, VA, USA, 220–229.

[29] Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. 2014. Probabilistic Programming. In *FOSE 2014: Future Soft. Eng. Proc.* ACM, New York, NY, USA, 167–181. https://doi.org/10.1145/2593882.2593900

[30] Steven Holtzen, Guy Van den Broeck, and Todd Millstein. 2020. Scaling Exact Inference for Discrete Probabilistic Programs. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 140 (Oct. 2020), 31 pages. https://doi.org/10.1145/3133904

[31] Frederick Jelinek. 1997. *Statistical Methods for Speech Recognition.* MIT Press, Cambridge, MA, USA.

[32] Charles Kemp, Joshua B. Tenenbaum, Thomas L. Griffiths, Takeshi Yamada, and Naonori Ueda. 2006. Learning Systems of Concepts with An Infinite Relational Model. In *AAAI 2006: Proc. 21st AAAI Conf. Artif. Intell.* AAAI Press, Palo Alto, CA, USA, 381–388.

[33] Daphne Koller and Nir Friedman. 2009. *Probabilistic Graphical Models: Principles and Techniques.* MIT Press, Cambridge, MA, USA.

[34] Christopher Krapu and Mark Borsuk. 2019. Probabilistic Programming: A Review for Environmental Modellers. *Environ. Model. Softw* 114 (April 2019), 40–48. https://doi.org/10.1016/j.envsoft.2019.01.014

[35] Tejas D. Kulkarni, Pushmeet Kohli, Joshua B. Tenenbaum, and Vikash Mansinghka. 2015. Picture: A Probabilistic Programming Language for Scene Perception. In *CVPR 2015: Proc. 2015 IEEE Conf. Comput. Vis.*

Pattern Recognit. IEEE Press, Piscataway, NJ, USA, 4390–4399.

[36] Jacob Laurel and Sasa Misailovic. 2020. Continualization of Probabilistic Programs with Correction. In *ESOP 2020: Proc. 29th Eur. Sympos. Program.* Springer, Cham, 366–393. https://doi.org/10.1007/978-3-030-44914-8_14

[37] Sang-Woo Lee, Christopher Watkins, and Byoung-Tak Zhang. 2014. Non-Parametric Bayesian Sum-Product Networks. In *LTPM 2014: Workshop Learn. Tractable Probab. Model.*

[38] Alexander K. Lew, Marco F. Cusumano-Towner, Benjamin Sherman, Michael Carbin, and Vikash K. Mansinghka. 2020. Trace Types and Denotational Semantics for Sound Programmable Inference in Probabilistic Languages. *Proc. ACM Program. Lang.* 4, POPL, Article 19 (Jan. 2020), 32 pages. https://doi.org/10.1145/3371087

[39] Vikash K. Mansinghka, Ulrich Schaechtle, Shivam Handa, Alexey Radul, Yutian Chen, and Martin Rinard. 2018. Probabilistic Programming with Programmable Inference. In *PLDI 2018: Proc. 39th ACM SIGPLAN Conf. Program. Lang. Des. Impl.* ACM, New York, NY, USA, 603–616. https://doi.org/10.1145/3192366.3192409

[40] Brian Milch, Bhaskara Marthi, Stuart Russell, David Sontag, Daniel L. Ong, and Andrey Kolobov. 2005. BLOG: Probabilistic Models with Unknown Objects. In *IJCAI 2005: Proc. 19th Int. Jt. Conf. Artif. Intell.* International Joint Conferences on Artificial Intelligence, Cham, 1352–1359.

[41] Alejandro Molina et al. 2020. SPFlow: An Easy and Extensible Library for Deep Probabilistic Learning using Sum-Product Networks. (Jan. 2020). arXiv:1901.03704

[42] Kevin Murphy and Mark A. Paskin. 2002. Linear-Time Inference in Hierarchical HMMs. In *NIPS 2001: Adv. Neural Inf. Proc. Syst. 14.* MIT Press, Cambridge, MA, USA, 833–840.

[43] Praveen Narayanan, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov. 2016. Probabilistic Inference by Program Transformation in Hakaru (System Description). In *FLOPS 2016: Proc. 13th Int. Sympos. Func. Log. Program.* Springer, Cham, 62–79. https://doi.org/10.1007/978-3-319-29604-3_5

[44] Praveen Narayanan and Chung-chieh Shan. 2020. Symbolic Disintegration with a Variety of Base Measures. *ACM Trans. Program. Lang. Syst.* 42, 2, Article 9 (May 2020), 60 pages. https://doi.org/10.1145/3374208

[45] Davide Nitti, Tinne De Laet, and Luc De Raedt. 2016. Probabilistic Logic Programing for Hybrid Relational Domains. *Mach. Learn.* 103 (April 2016), 407–449. https://doi.org/10.1007/s10994-016-5558-8

[46] Aditya Nori, Chung-Kil Hur, Sriram Rajamani, and Selva Samuel. 2014. R2: An Efficient MCMC Sampler for Probabilistic Programs. In *AAAI 2014: Proc. 28th AAAI Conf. Artif. Intell.* AAAI Press, 2476–2482.

[47] Hoifung Poon and Pedro Domingos. 2011. Sum-Product Networks: A New Deep Architecture. In *UAI 2011: Proc. 27th Annu. Conf. Uncertain. Artif. Intell.* AUAI Press, Arlington, VA, USA, 337–346.

[48] Fabrizio Riguzzi. 2018. *Foundations of Probabilistic Logic Programming: Languages, Semantics, Inference and Learning.* River Publishers, Delft, The Netherlands.

[49] Feras Saad, Leonardo Casarsa, and Vikash Mansinghka. 2017. Probabilistic Search For Structured Data Via Probabilistic Programming and Nonparametric Bayes. (April 2017). arXiv:1704.01087

[50] Feras Saad and Vikash Mansinghka. 2016. Probabilistic Data Analysis with Probabilistic Programming. (Aug. 2016). arXiv:1608.05347

[51] Feras Saad and Vikash Mansinghka. 2016. A Probabilistic Programming Approach to Probabilistic Data Analysis. In *NIPS 2016: Adv. Neural Inf. Proc. Syst. 29.* Curran Associates, Inc., Red Hook, NY, USA, 2011–2019.

[52] Feras Saad and Vikash Mansinghka. 2017. Detecting Dependencies in Sparse, Multivariate Databases using Probabilistic Programming and Non-parametric Bayes. In *AISTATS 2017: Proc. 20th Int. Conf. Artif. Intell. Stat.* PMLR, 632–641.

[53] Feras A. Saad, Marco F. Cusumano-Towner, Ulrich Schaechtle, Martin C. Rinard, and Vikash K. Mansinghka. 2019. Bayesian Synthesis of Probabilistic Programs for Automatic Data Modeling. *Proc. ACM Program. Lang.* 3, POPL, Article 36 (Jan. 2019), 32 pages. https://doi.org/10.1145/3290350

[54] Feras A. Saad and Vikash K. Mansinghka. 2018. Temporally-reweighted Chinese Restaurant Process Mixtures for Clustering, Imputing, and Forecasting Multivariate Time Series. In *AISTATS 2018: Proc. 21st Int. Conf. Artif. Intell. Stat.* PMLR, 755–764.

[55] Sriram Sankaranarayanan, Aleksandar Chakarov, and Sumit Gulwani. 2013. Static Analysis for Probabilistic Programs: Inferring Whole Program Properties from Finitely Many Paths. In *PLDI 2013: Proc. 34th ACM SIGPLAN Conf. Program. Lang. Des. Impl.* ACM, New York, NY, USA, 447–458. https://doi.org/10.1145/2491956.2462179

[56] David J. Spiegelhalter, A. Philip Dawid, Steffen L. Lauritzen, and Robert G. Cowell. 1993. Bayesian Analysis in Expert Systems. *Statist. Sci.* 8, 3 (Aug. 1993), 219–247. https://doi.org/10.1214/ss/1177010888

[57] Siddharth Srivastava, Nicholas Hay, Yi Wu, and Stuart Russell. 2017. The Extended Semantics for Probabilistic Programming Languages. In *PPS 2017: Workshop Probab. Program. Semant.*

[58] Andreas Stuhlmüller and Noah Goodman. 2012. A Dynamic Programming Algorithm for Inference in Recursive Probabilistic Programs. (Sept. 2012). arXiv:1206.3555

[59] Andrew Thomas. 1994. BUGS: A Statical Modelling Package. *RTA/BCS Modul. Lang. Newsl.* 2 (1994), 36–38.

[60] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. 2005. *Probabilistic Robotics.* MIT Press, Cambridge, MA, USA.

[61] Neil Toronto, Jay McCarthy, and David Van Horn. 2015. Running Probabilistic Programs Backwards. In *ESOP 2015: Proc. 24th Eur. Sympos. Program.* Springer, Berlin, 53–79. https://doi.org/10.1007/978-3-662-46669-8_3

[62] Martin Trapp, Robert Peharz, Hong Ge, Franz Pernkopf, and Zoubin Ghahramani. 2019. Bayesian Learning of Sum-Product Networks. In *NeurIPS 2019: Adv. Neural Inf. Proc. Syst. 32.* Curran Associates, Inc., Red Hook, NY, USA, 6347–6358.

[63] F. Turbak, D. Gifford, and M.A. Sheldon. 2008. *Design Concepts in Programming Languages.* MIT Press, Cambridge, MA, USA.

[64] Antonio Vergari. 2021. *A Systematic View of the Literature on Probabilistic Circuits, Computational Graphs Encoding Tractable Probability Distributions.* Retrieved Mar 7, 2021 from https://arranger1044.github.io/probabilistic-circuits/

[65] Antonio Vergari, Alejandro Molina, Robert Peharz, Zoubin Ghahramani, Kristian Kersting, and Isabel Valera. 2019. Automatic Bayesian Density Analysis. In *AAAI 2019: Proc. 23rd AAAI Conf. Artif. Intell.* AAAI Press, Palo Alto, CA, USA, 5207–5215.

[66] Jonas Vlasselaer, Guy Van den Broeck, Angelika Kimmig, Wannes Meert, and Luc De Raedt. 2015. Anytime Inference in Probabilistic Logic Programs with Tp-Compilation. In *IJCAI 2015: Proc. 24th Int. Jt. Conf. Artif. Intell.* International Joint Conferences on Artificial Intelligence, Cham, 1852–1858.

[67] David Wingate and Theophane Weber. 2013. Automated Variational Inference in Probabilistic Programming. (Jan. 2013). arXiv:1301.1299

[68] Yi Wu, Siddharth Srivastava, Nicholas Hay, Simon Du, and Stuart Russell. 2018. Discrete-Continuous Mixtures in Probabilistic Programming: Generalized Semantics and Inference Algorithms. In *ICML 2018: Proc. 35th Int. Conf. Mach. Learn.* PMLR, 5343–5352.

[69] Jieyuan Zhang and Jingling Xue. 2019. Incremental Precision-Preserving Symbolic Inference for Probabilistic Programs. In *PLDI 2019: Proc. 40th ACM SIGPLAN Conf. Program. Lang. Des. Impl.* ACM, New York, NY, USA, 237–252. https://doi.org/10.1145/3314221.3314623

[70] Pedro Zuidberg Dos Martires, Anton Dries, and Luc De Raedt. 2019. Exact and Approximate Weighted Model Integration with Probability Density Functions Using Knowledge Compilation. In *AAAI 2019: Proc. 23rd AAAI Conf. Artif. Intell.* AAAI Press, Palo Alto, CA, USA, 7825–7833.