Knowledge Compilation with Discrete and Continuous Variables

Theodore Long[1] **Alexander K. Lew**[1]

[1]Yale University, New Haven, Connecticut, USA

Abstract

We present a new technique for exact inference in hybrid probabilistic programs, those that make both discrete and continuous random choices. Our approach extends knowledge compilation, a popular technique for discrete inference, to support conjugate continuous latent variables and measure-zero observations. We implement our approach in the Hybrid Pluck PPL, and find that it yields significant speedups relative to existing systems for exact inference in hybrid models.

1 INTRODUCTION

In general, inference in probabilistic programs is intractable. But we might hope that when a probabilistic program happens to admit tractable marginalization or conditioning—e.g., due to independencies, conjugate relationships, or low-dimensional latent spaces—automated tooling could exploit this structure and provide fast, correct inference.

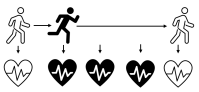
For example, consider the program in Fig. 1. Its goal is to detect whether a user is running or walking, based on regular readings from a heart rate monitor. A priori, we do not know the user's typical heart rate, nor do we know their current mode of exercise (purely walking, purely running, switching at intervals, etc.). To model this, we place broad Gaussian priors over the user's baseline heart rate and the amount it increases when running. We model the user's exercise as a hidden Markov model, with Boolean hidden state (running or not) and noisy heart rate observations. We place Beta priors on the frequencies p_{run} and p_{walk} with which the user switches from running to walking and vice versa.

Using the conjugacy of the Beta and Bernoulli distributions, together with the closure of Gaussian random variables under addition, this model's marginal and posterior distributions can be computed in closed form. Despite this fact,

```
# ACTIVITY DETECTION
base, incr = normal(70, 20), normal(50, 5)
p_run, p_walk = beta(1, 1), beta(1, 1)

running = flip(0.5)
for t in range(T):
  if running:
    hr = normal(base + incr, 5)
    running = not(flip(p_walk))
  else:
    hr = normal(base, 5)
    running = flip(p_run)
  observe(hr == observed_hrs[t])

return running, base
```



Readings:
[82.0, 81.0, 80.0,
83.0, 130.0, 134.0,
130.0, 105.0]

Figure 1: A hidden Markov model adapted from Li et al. [2024], but with unknown parameters. Based on heart rate readings, the goal is to infer if the user is walking or running and parameters such as baseline walking heart rate. Our system performs exact inference in this model in <10ms.

existing algorithms for exact inference in probabilistic programs cannot handle this model efficiently. Dice, Roulette, and Pluck [Holtzen et al., 2020, Moy et al., 2025, Bowers et al., 2025] do not handle continuous variables at all; MaPPL and SPPL [Li et al., 2024, Saad et al., 2021] handle them but not as latents; and PSI [Gehr et al., 2020] accepts the program but times out during inference.

This work develops a new approach to automating exact inference for a broad class of tractable probabilistic programs. Our key contribution is an extension of the widely used *knowledge compilation* technique for discrete inference (which powers Dice, Roulette, and Pluck [Holtzen

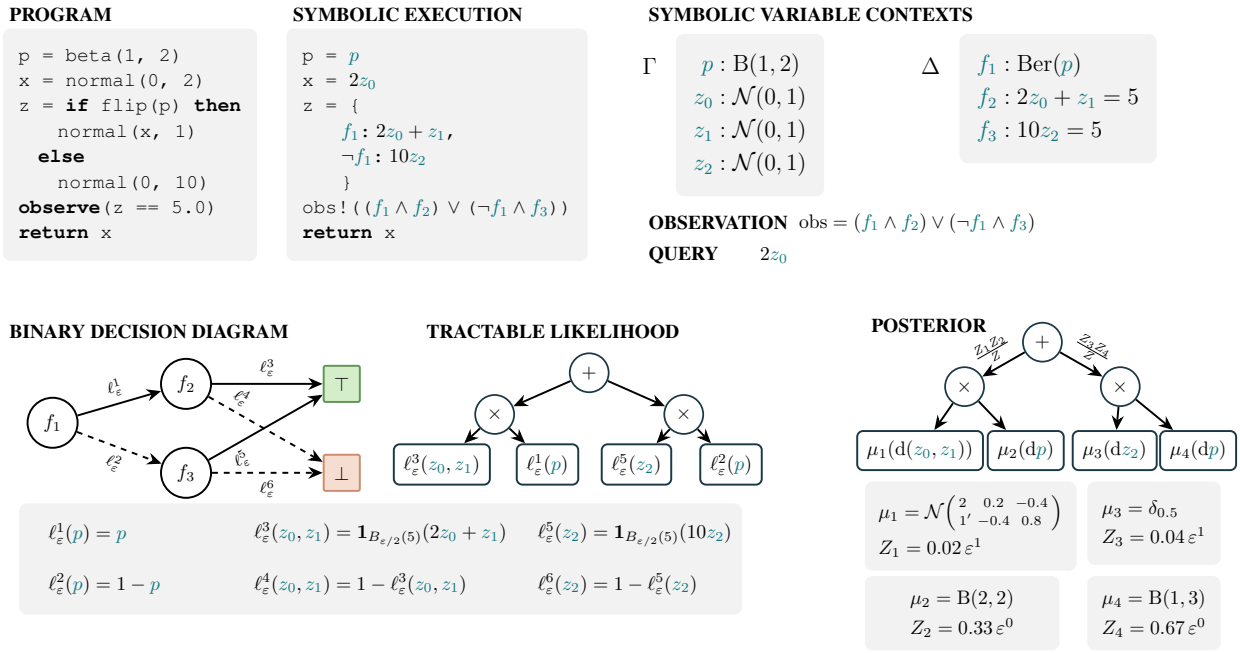


Figure 2: Overview of our approach. A program (top left) is first *symbolically executed* (§2.1) to derive representations of: the prior on continuous variables (Γ), the conditional distributions of discrete variables (Δ), the observation predicate (obs), and the query. The observation predicate is compiled to a weighted binary decision diagram, with weights from a semiring of *tractable likelihoods*. A weighted model count computes a sum-product representation of the overall likelihood function (§2.2). Finally, the prior Γ is combined with the likelihood to yield a sum-product network encoding the posterior (§2.3).

et al., 2020, Moy et al., 2025, Bowers et al., 2025]) to support a richer set of language features, including conjugate continuous priors (Beta-Bernoulli, Dirichlet-Categorical, Gaussian-Gaussian, Gamma-Poisson, and others) and measure zero observations (§2). In preliminary experiments, our algorithm appears to be more efficient than approaches based on computer algebra [Gehr et al., 2020, Martires et al., 2019] or discretization [Garg et al., 2024], answering queries orders of magnitude faster than existing systems (§3).

2 OUR APPROACH

Our approach is illustrated in Fig. 2. We now walk through each stage of the inference pipeline: symbolic execution of the user program (§2.1), compilation of a tractable likelihood (§2.2), and finally derivation of the posterior (§2.3).

2.1 SYMBOLIC EXECUTION

Given a probabilistic program, our first step is to perform *symbolic execution* to build a symbolic representation of the prior, observations, and query. Symbolic execution runs the program, except that when random sampling operations are encountered, instead of drawing random samples, fresh *symbolic* random variables are deterministically generated, and recorded into a running *context*, along with information about their types and distributions. In particular, we maintain *two* symbolic contexts: Γ for continuous variables, and Δ for discrete variables.

Continuous variables. The continuous-variable context Γ records a list of symbolic continuous variable names, each associated with a prior distribution. The priors for continuous variables in Γ are required to be *independent*. Because program variables are often *not* independent, the symbolic continuous variables in Γ are *not* necessarily in one-to-one correspondence with user-defined program variables. For example, in Fig. 2, the program variables p, x, and z are correlated. Symbolic execution represents them as more complex symbolic expressions over four independent random variables, p, z_0, z_1, z_2, in Γ. This is achieved via a handful of local rules. For example, $\mathsf{normal}(e_1, e_2)$ evaluates e_1 to a symbolic mean μ and e_2 to a symbolic standard deviation σ, then adds a new *standard normal* variable $z : \mathcal{N}(0, 1)$ to Γ and returns the symbolic value $\sigma \cdot z + \mu$.

Discrete variables. Boolean random choices (as well as Booleans that depend deterministically on continuous random choices) are stored in Δ. The discrete symbolic variables in Δ are required to be *conditionally* independent, given the continuous variables in Γ. Deterministic predicates on Γ, e.g. $2z_0 + z_1 = 5$, are constant given Γ and therefore trivially (conditionally) independent of all other entries of Δ. Coin flips in a program may fail to be in-

dependent, but symbolic execution re-expresses them as functions of independent flips. For example, symbolically evaluating $\mathsf{x} = \mathsf{flip}(\mathsf{if}\ \mathsf{flip}(0.5)\ \mathsf{then}\ p\ \mathsf{else}\ 0.3)$ adds three symbolic Booleans to the context: $\Delta = f_0 : \mathsf{Ber}(0.5), f_1 : \mathsf{Ber}(p), f_2 : \mathsf{Ber}(0.3)$. It then sets the program variable x to the symbolic formula $(f_0 \wedge f_1) \vee (\neg f_0 \wedge f_2)$.

Interpreting other program constructs. Each primitive in the language is extended to operate on symbolic values, which represent values that are dependent on the random variables in Γ and Δ. When branching on a symbolic Boolean (i.e., a Boolean formula ϕ over the variables in Δ), we symbolically execute both branches, then *merge* the results using ϕ. For example, the merge of two formulae ϕ_1 and ϕ_2 under branch condition ϕ is the formula $\phi \wedge \phi_1 \vee \neg\phi \wedge \phi_2$. The merge of two symbolic numeric expressions e_1, e_2 is a *symbolic union* of the form $\{\phi : e_1, \neg\phi : e_2\}$. The observe construct, given a symbolic Boolean ϕ, updates a running *obs* formula (initialized to $\top$) to equal *obs* $\wedge\ \phi$. When the program returns, we have accumulated a continuous context Γ, a discrete context Δ (denoting a conditional distribution on $\{0,1\}^{|\Delta|}$, given the continuous variables), a Boolean formula *obs* over Δ encoding a condition, and a symbolic value returned by the program, called the query.

2.2 WEIGHTED MODEL COUNTING IN THE SEMIRING OF LIKELIHOODS

The next phase of our algorithm generates a symbolic *likelihood function*: a function that, given particular values θ for the variables in Γ, computes the conditional probability (under the conditional distribution specified by Δ) that *obs* holds. We can cast this as an instance of the well-known *weighted model counting* problem: given a Boolean formula φ over the finite collection of Boolean variables Δ, and a map w specifying a *weight* $w(x)$ and *negative weight* $w(\neg x)$ for each variable $x \in \Delta$, the weighted model count is

$$\mathrm{WMC}(\varphi, w) = \sum_{\sigma \models \varphi} \prod_{x \in \Delta} w(x)^{\sigma(x)} w(\neg x)^{1-\sigma(x)} \quad (1)$$

where the sum is over *assignments* $\sigma : \Delta \to \{0,1\}$ that satisfy φ, and the product is over the variables in Δ.

Although the weights in weighted model counting are typically taken to be probabilities or real numbers, the definition makes sense (and the standard algorithms work) for weights from any semiring. We choose to work in a semiring of *likelihood functions* $\ell_\epsilon : \Theta \to [0,1]$, where Θ is the set of assignments θ specifying a real number $\theta(z)$ for each $z \in \Gamma$.[1] These likelihoods form a semiring under pointwise addition and multiplication. For each variable x in Δ, we set $w(x)$ to the likelihood function $\ell^x_\epsilon(\theta) = \mathbb{P}(x = \top \mid \theta)$, and $w(\neg x) = \ell^{\neg x}_\epsilon(\theta) = \mathbb{P}(x = \bot \mid \theta)$. For example, suppose $\Delta = f_1 : \mathsf{Ber}(p), f_2 : 2z_0 = 5$. Then $\ell^{f_1}_\epsilon(\theta) = \theta(p)$ and $\ell^{f_2}_\epsilon(\theta) = 1_{[5-\epsilon/2, 5+\epsilon/2]}(2\theta(z_0))$.

Proposition 1. *Letting* $\ell_\epsilon = \mathit{WMC}(\mathit{obs}, w)$,

$$\mathbb{P}(\mathit{obs} = \top \mid \theta) = \ell_0(\theta).$$

Concrete representation. Concretely, we represent likelihood functions as *sum-product networks* [Poon and Domingos, 2012]. *Leaf* nodes are labeled with symbolic representations of particular tractable likelihood functions. For example, $\mathsf{BetaBern}(m; p, n)$, where p is a Beta-distributed variable from Γ, represents the likelihood function $\theta \mapsto \theta(p)^m(1-\theta(p))^{n-m}$. A *Sum* node represents the pointwise sum of the likelihood functions represented by its children. A *Product* node must have children with disjoint *scopes*, where the scope of a likelihood function is the subset of variables in Γ that it depends on. Product nodes represent the pointwise products of their children. Likelihood functions represented as SPNs can be efficiently added and multiplied;[2] to perform our weighted model count, we initialize the weight map w to contain appropriate Leaf nodes, then run a standard algorithm for WMC, which compiles the *obs* formula to a binary decision diagram and then computes the model count in a single bottom-up pass over the diagram. An example BDD and the SPN resulting from this procedure are shown in the bottom left panel of Fig. 2.

2.3 INFERENCE WITH EXACT OBSERVATIONS

Given our likelihood SPN representing $\ell_\epsilon(\theta)$, and the factorized prior $p(\theta) = \prod_{x \in \Gamma} p_x(\theta(x))$ encoded by the context Γ, our goal is now to compute the marginal likelihood Z and the posterior μ, defined as

$$Z = \lim_{\epsilon \to 0} \frac{1}{\epsilon^n} \int_\Theta \ell_\epsilon(\theta) p(\theta) d\theta \quad (2)$$

$$\mu = \operatorname*{w-lim}_{\epsilon \to 0} \frac{\ell_\epsilon \odot p}{\int_\Theta \ell_\epsilon(\theta) p(\theta) d\theta} \quad (3)$$

$$n = \min_{\mathbb{Z}_{\geq 0}} \left\{ k \;\middle|\; \lim_{\epsilon \to 0} \int_\Theta \frac{\ell_\epsilon(\theta) p(\theta)}{\epsilon^k} d\theta > 0 \right\} \quad (4)$$

where the limit in Eq. 3 is the weak limit of measures.

These limits are necessary to handle the fact that, when $\epsilon = 0$, the event *obs* may have measure zero. Inspired by the approach of Jacobs [2021], we write $Z = a\epsilon^n$ to mean that the marginal likelihood in Eq. 2 is a and the integer in Eq. 4 is n. Values of the form $a\epsilon^n$ form a semiring $\mathbb{R}[\epsilon_{\min}]$,

[1] Our likelihoods are parameterized by a *tolerance parameter* $\epsilon > 0$ that we write in the subscript; ϵ controls how exact our equality tests are.

[2] This relies on efficient products of leaves. For example,

$$\mathsf{BetaBern}(m;p,n)\mathsf{BetaBern}(m';p,n') = \mathsf{BetaBern}(m+m';p,n+n')$$

with $a\epsilon^n \cdot b\epsilon^m = ab\epsilon^{n+m}$, $a\epsilon^n + b\epsilon^n = (a+b)\epsilon^n$, and $a\epsilon^n + b\epsilon^m = a\epsilon^n$ when $n < m$.

Both μ and Z can be computed compositionally via one bottom-up pass on the sum-product likelihood. *Leaf nodes* represent simple, conjugate likelihoods for which we assume closed forms exist for the posterior and marginal likelihood. For example, for the likelihood $\mathsf{BetaBern}(m; p, n)$, where $p : B(a,b) \in \Gamma$, we have $Z = \frac{B(a+m,b+n-m)}{B(a,b)}\epsilon^0$, and the posterior is $\mu = \mathrm{Beta}(a+m, b+n-m)$. To handle product and sum nodes, we rely on the following two facts:

Proposition 2 (Product nodes)**.** *Suppose Γ is partitioned into disjoint subsets $S_1, \ldots, S_k$, and the likelihood factorizes as $\ell_\epsilon(\theta) = \prod_{i=1}^k \ell_\epsilon^i(\theta_{S_i})$. Let $Z_i = a_i\epsilon^{n_i}$ and μ_i denote the marginal likelihood and posterior of child i. Then the marginal likelihood and posterior of ℓ_ϵ are*

$$Z = \prod_{i=1}^k Z_i = \left(\prod_{i=1}^k a_i\right)\epsilon^{\sum_{i=1}^k n_i}, \qquad \mu = \bigotimes_{i=1}^k \mu_i,$$

where $\bigotimes$ denotes the independent product measure.

Proposition 3 (Sum nodes)**.** *Suppose $\ell_\epsilon(\theta) = \sum_{i=1}^k \ell_\epsilon^i(\theta)$, and let $Z_i = a_i\epsilon^{n_i}$ and μ_i denote the marginal likelihood and posterior of each child ℓ_ϵ^i with respect to the prior p. Then the marginal likelihood and posterior of ℓ_ϵ are*

$$Z = \sum_{i=1}^k Z_i, \qquad \mu = \sum_{i=1}^k \frac{Z_i}{Z}\,\mu_i,$$

where the sum $Z = \sum_i Z_i$ is taken in $\mathbb{R}[\epsilon_{\min}]$: writing $n = \min_i n_i$, we have $Z = \left(\sum_{i:\, n_i = n} a_i\right)\epsilon^n$, with only the children with $n_i = n$ contributing to the mixture.

When traversing the likelihood SPN bottom-up, at each node we produce either a closed-form posterior distribution, an independent product of closed-form posteriors on disjoint projections of the full space, or a weighted mixture of tractable posteriors. Therefore, we once again recover the structure of an SPN, and our method represents the intermediate and final posteriors μ_i, μ as SPNs (Fig. 2, bottom right). This allows us to easily generate samples or marginalize variables from our posterior μ.

3 RESULTS

We have implemented this approach in Hybrid Pluck, an extension to the existing Pluck PPL [Bowers et al., 2025], which performs a *lazy* variant of symbolic execution. We compare the runtime to other systems for both exact and approximate inference on a number of mixed discrete-continuous models.

Table 1 shows a comparison to `PSI` [Gehr et al., 2020], a system that performs symbolic exact inference on hybrid models, and `HyBit` [Garg et al., 2024], a system which extends knowledge compilation to support continuous variables through the use of *discrete approximations*. Since `HyBit` can only approximate continuous variables, we report both the runtime and the approximation error as a relative error of some posterior quantity (an expectation or event probability). Our implementation outperforms both `PSI` and `HyBit` on all benchmark programs, as well as providing exact inference.

Table 1: Hybrid Pluck vs. exact inference in PSI and approximate (bit-blasted) inference (HyBit), with relative approximation error. Details of experimental procedure can be found in appendix A.

	Inference time (s)			Error (%)
Program	Hybrid Pluck	PSI	HyBit	Hybit
GPA	$\mathbf{1.64\times10^{-3}}$	9.15×10^{-1}	1.71×10^{-1}	0.00
clickGraph	$\mathbf{5.64\times10^{-4}}$	3.22×10^{0}	4.80×10^{-1}	2.28
clinicalTrial1	$\mathbf{3.59\times10^{-2}}$	T/O	3.83×10^{0}	0.14
clinicalTrial2	$\mathbf{2.80\times10^{-4}}$	4.38×10^{-1}	2.41×10^{-1}	0.06
coinBias	$\mathbf{1.72\times10^{-4}}$	2.01×10^{-1}	4.49×10^{-2}	0.20
conj. gaussians	$\mathbf{1.21\times10^{-4}}$	1.80×10^{-2}	1.07×10^{0}	0.07
normal mixture	$\mathbf{2.89\times10^{0}}$	T/O	4.35×10^{1}	0.01
polyreg	$\mathbf{1.05\times10^{-2}}$	1.11×10^{2}	T/O	—
run walk	$\mathbf{6.15\times10^{-3}}$	T/O	T/O	—
soccer	$\mathbf{7.10\times10^{0}}$	T/O	—	—
spacex	$\mathbf{3.81\times10^{-4}}$	3.12×10^{-2}	4.26×10^{1}	0.0041
weekend	$\mathbf{4.96\times10^{-4}}$	7.60×10^{-2}	2.17×10^{1}	4.03

T/O = timeout (300s) "—": not applicable (cannot express, or returns error).

4 CONCLUSION

We have presented a method for incorporating continuous latent variables and exact continuous observations into existing discrete inference methods based on knowledge compilation. We implement our method as an extension to an existing PPL for discrete inference. Our experiments demonstrate the empirical improvements in runtime of our method, compared to other systems for both exact and approximate inference in models with mixed discrete and continuous variables.

In many hybrid models, exact inference still scales poorly with dataset size. However, the ability to do exact inference in flexibly specified models opens up new avenues for automating approximate inference techniques. As a step toward this goal, we have implemented a Gibbs feature, which uses the exact inference machinery described here, as well as sampling algorithms for SPNs, to evaluate and sample the conditional densities required by block Gibbs sampling. We hope to further explore how our method can be integrated into systems for approximate inference, using exact subproblem inference to scale to larger problems.

Acknowledgements

We thank the reviewers for their comments and suggestions, which were valuable in improving the presentation and highlighting additional relevant work.

References

Maddy Bowers, Alexander K. Lew, Joshua B. Tenenbaum, Armando Solar-Lezama, and Vikash K. Mansinghka. Stochastic Lazy Knowledge Compilation for Inference in Discrete Probabilistic Programs. *Proceedings of the ACM on Programming Languages*, 9(PLDI):222:1863–222:1887, June 2025. doi: 10.1145/3729325.

Poorva Garg, Steven Holtzen, Guy Van den Broeck, and Todd Millstein. Bit Blasting Probabilistic Programs. *Proceedings of the ACM on Programming Languages*, 8(PLDI):182:865–182:888, June 2024. doi: 10.1145/3656412.

Timon Gehr, Samuel Steffen, and Martin Vechev. λPSI: Exact inference for higher-order probabilistic programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, pages 883–897, New York, NY, USA, June 2020. Association for Computing Machinery. ISBN 978-1-4503-7613-6. doi: 10.1145/3385412.3386006.

Steven Holtzen, Guy Van den Broeck, and Todd Millstein. Scaling Exact Inference for Discrete Probabilistic Programs. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–31, November 2020. ISSN 2475-1421. doi: 10.1145/3428208.

Jules Jacobs. Paradoxes of probabilistic programming: And how to condition on events of measure zero with infinitesimal probabilities. *Proc. ACM Program. Lang.*, 5(POPL): 58:1–58:26, January 2021. doi: 10.1145/3434339.

Jianlin Li, Eric Wang, and Yizhou Zhang. Compiling Probabilistic Programs for Variable Elimination with Information Flow. *Proceedings of the ACM on Programming Languages*, 8(PLDI):218:1755–218:1780, June 2024. doi: 10.1145/3656448.

Pedro Zuidberg Dos Martires, Anton Dries, and Luc De Raedt. Exact and Approximate Weighted Model Integration with Probability Density Functions Using Knowledge Compilation. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(01):7825–7833, July 2019. ISSN 2374-3468. doi: 10.1609/aaai.v33i01.33017825.

Cameron Moy, Jack Czenszak, John M. Li, Brianna Marshall, and Steven Holtzen. Roulette: A Language for Expressive, Exact, and Efficient Discrete Probabilistic Programming. *Proceedings of the ACM on Programming Languages*, 9(PLDI):2081–2105, June 2025. ISSN 2475-1421. doi: 10.1145/3729334.

Hoifung Poon and Pedro Domingos. Sum-Product Networks: A New Deep Architecture, February 2012.

Feras A. Saad, Martin C. Rinard, and Vikash K. Mansinghka. SPPL: Probabilistic programming with fast exact symbolic inference. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pages 804–819, New York, NY, USA, June 2021. Association for Computing Machinery. ISBN 978-1-4503-8391-2. doi: 10.1145/3453483.3454078.

Knowledge Compilation with Discrete and Continuous Variables (Supplementary Material)

Theodore Long[1] **Alexander K. Lew**[1]

[1]Yale University, New Haven, Connecticut, USA

A EXPERIMENTAL DETAILS

All results shown in Table 1 were measured on a single Intel Emerald Rapids CPU core with 64GB of memory. The times shown *exclude* fixed startup costs, such as runtime loading or JIT compilation. Runs that did not complete within a 300-second timeout are marked T/O in the table. The relative error metric is computed based on a single scalar output quantity which is produced for each program, with the value generated by Hybrid Pluck used as ground truth. The soccer program is not expressible in HyBit due to Dirichlet random variables not being supported.

B SUM-PRODUCT NETWORKS

Sum-Product Networks (SPNs) are an example of probabilistic circuits, a class of graphical models specifying a joint distribution over several variables with a particular structure that allows for tractable probabilistic queries such as conditioning, marginalization, and sampling.

Our system uses SPNs to represent both likelihoods and posterior distributions, and may depart in certain details from other treatments. We provide a formal definition encompassing both representations, as well as a description of the semiring operations for the semiring of likelihood SPNs.

B.1 DEFINITION

A sum-product network consists of a set of `Leaf`, `Sum`, and `Product` nodes, where each node has a weight w in some semiring R and set S known as the *scope*, which contains zero or more random variables. Each `Leaf` node contains associated data representing either a likelihood or a distribution, while `Sum` and `Product` nodes contain references to a set of child nodes. The scopes of `Sum` and `Product` nodes are defined as the union of the scopes of their children.

SPNs must also satisfy the following conditions:

Decomposability. For each `Product` node, the scopes of its children must be disjoint.

Weight Normalization. The sum of the weights of a `Sum` node must be 1, and all children of a `Product` node must have weight 1 (the overall normalization constant gets 'pushed' to the outer node).

Flattening. The children of `Sum` nodes must be either `Product` or `Leaf` nodes, and the children of `Product` nodes must be either `Sum` or `Leaf` nodes.

Many definitions of SPNs also require 'completeness' or 'smoothness', which requires that the children of sum nodes have the *same* scope (whereas we allow them to be subsets of the parent node's scope). This is not desirable in our setting - the likelihood SPN may only reference a small subset of variables in any given branch, so it is desirable to keep the representation compact rather than inserting 'dummy' likelihoods that evaluate to 1 and reference all variables not already captured in the scope.

We note that rather than considering separate weights $w \in R$, we equivalently view weights as `Leaf` nodes with an empty scope, and weighted nodes as a product of a node and a weight Leaf node.

B.2 SEMIRING OPERATIONS

We assume the existence of an associative, commutative product operation $\star_L$ on each of the different types L of `Leaf` nodes. For details of the different types $(L, \star_L)$ see appendix C. We denote real-valued weights (or equivalently, leaves with empty scope) with a real number r, s while leaves of type L, K have scopes S, T and data λ, κ. $\texttt{Node}(S)$ is used to represent a node of arbitrary type with scope S. We write $w \cdot N$ for the node N carrying weight w (equivalently, following the convention above, the product of N with the weight leaf w), and for a set C of weighted children and a scalar a we write $a\,C \coloneqq \{\, (aw) \cdot c \mid (w \cdot c) \in C \,\}$ for the rescaling of every child weight by a. Each variable is associated with a single leaf type, so that any two leaves with overlapping scope are necessarily of the same type L; such leaves combine via $\star_L$ even when their scopes overlap only partially, yielding a single type-L leaf over the union of their scopes. The set of likelihood SPNs forms a semiring with distinguished elements 0_{SPN}, 1_{SPN} and operations $+, \star$ defined as follows:

$$0_{\text{SPN}} \coloneqq 0 \in \mathbb{R} \tag{5}$$

$$1_{\text{SPN}} \coloneqq 1 \in \mathbb{R} \tag{6}$$

$$r + \texttt{Leaf}\big(S,\ \lambda\big) \coloneqq (r+1) \cdot \texttt{Sum}\big(\{\tfrac{r}{r+1},\ \tfrac{1}{r+1} \cdot \texttt{Leaf}\big(S,\ \lambda\big)\}\big) \tag{7}$$

$$\texttt{Leaf}\big(S,\ \lambda\big) + \texttt{Leaf}\big(T,\ \kappa\big) \coloneqq 2 \cdot \texttt{Sum}\big(\{\tfrac{1}{2} \cdot \texttt{Leaf}\big(S,\ \lambda\big),\ \tfrac{1}{2} \cdot \texttt{Leaf}\big(T,\ \kappa\big)\}\big) \tag{8}$$

$$r \cdot N + s \cdot M \coloneqq (r+s) \cdot \texttt{Sum}\big(\{\tfrac{r}{r+s} \cdot N,\ \tfrac{s}{r+s} \cdot M\}\big) \tag{9}$$

$$r \cdot \texttt{Sum}\big(C\big) + s \cdot M \coloneqq (r+s) \cdot \texttt{Sum}\big(\tfrac{r}{r+s}\, C \cup \{\tfrac{s}{r+s} \cdot M\}\big) \tag{10}$$

$$r \cdot \texttt{Sum}\big(C_1\big) + s \cdot \texttt{Sum}\big(C_2\big) \coloneqq (r+s) \cdot \texttt{Sum}\big(\tfrac{r}{r+s}\, C_1 \cup \tfrac{s}{r+s}\, C_2\big) \tag{11}$$

$$\texttt{Leaf}\big(S,\ \lambda\big) \star \texttt{Leaf}\big(T,\ \lambda'\big) \coloneqq \texttt{Leaf}\big(S \cup T,\ \lambda \star_L \lambda'\big) \qquad (S \cap T \neq \emptyset) \tag{12}$$

$$r \cdot N(S) \star s \cdot N'(T) \coloneqq (rs) \cdot \texttt{Product}\big(\{N(S),\, N'(T)\}\big) \qquad (S \cap T = \emptyset) \tag{13}$$

$$r \cdot N(S) \star s \cdot N'(T) \coloneqq (rs) \cdot \texttt{recursive\_product}(N(S),\, N'(T)) \qquad (S \cap T \neq \emptyset) \tag{14}$$

Throughout, N and M range over `Leaf` and `Product` nodes, λ, λ' are both of type L, and the rules are applied up to commutativity, matching the most specific applicable rule first. Each $+$ rule renormalizes the combined children so that their weights again sum to 1, pushing the total weight $r + s$ out to the enclosing node; the disjoint-scope $\star$ rule likewise pushes the product rs of the two weights outward. The same-type leaf rule applies whenever the two leaves' scopes overlap; two leaves with disjoint scopes instead fall under the disjoint-scope rule. A bare `Leaf` or `Product` node can be viewed as a single-child `Sum`, so the second and third $+$ rules are the special cases of the third in which one or both operands have a single child, and flattening is preserved because no `Sum` is ever nested directly beneath another.

The first two $\star$ rules and the disjoint-scope rule reduce every product to one of the two base cases (two overlapping leaves of the same type, or two nodes with disjoint scopes) *unless* the operands are higher-level nodes that share variables. In that case the overlapping-scope rule defers to `recursive_product` (Algorithm 1), which decomposes the operands until reaching a base case. It distributes the product over the children of a `Sum` node, and partitions the factors of `Product` nodes (treating a bare leaf as its own single factor) into scope-connected groups, recombining each group using the semiring product $\star$, which dispatches back to the base cases or recurses. We write $\bigstar$ for the iterated product and $\sum$ for the iterated sum.

Within a group the factors are connected by overlap, so recombining them with $\star$ either reaches the same-type leaf base case or recurses on strictly smaller nodes; across groups the group results have disjoint scopes, so $\star$ yields a `Product` node. Since each recursive $\star$ call acts on strictly smaller nodes, the recursion terminates, and the fact that overlapping leaves share a type rules out the only case in which two leaves could fail to combine.

Note that in our implementation semiring operations use logarithmic weights for numerical stability.

Algorithm 1 `recursive_product`(A, B) for nodes with overlapping scopes

Require: Weight-1 nodes A, B with $\text{scope}(A) \cap \text{scope}(B) \neq \emptyset$
Ensure: A normalized SPN node representing $A \star B$

if $A = \texttt{Sum}\big(\{ w_i \cdot c_i \}_i\big)$ **then**
  return $\sum_i w_i \cdot (c_i \star B)$ ▷ distribute over the mixture; + renormalizes
else if B is a `Sum` node **then**
  return `recursive_product`(B, A) ▷ symmetric
else ▷ A, B are `Leaf`/`Product` nodes
  $F \leftarrow \text{factors}(A) \cup \text{factors}(B)$ ▷ factors of a leaf is the leaf itself
  Partition F into groups $G_1, \ldots, G_p$ where f, f' share a group iff their scopes are linked by a chain of pairwise-overlapping factors
  return $\bigstar_{k=1}^{p} \big(\bigstar_{f \in G_k} f \big)$ ▷ within a group: overlap ⇒ recurse; across groups: disjoint ⇒ `Product`
end if

C REPRESENTATIONS OF CONJUGATE FAMILIES

In appendix B.2 we assumed the existence of various types of leaf nodes representing either likelihoods or posterior distributions, as well as a product operation $\star_L$ for each type L of likelihood leaf node. In this section we describe all the leaf node types currently implemented in our framework, although we note that in principle any conjugate family should be representable.

C.1 BETA-BERNOULLI

Given a beta-distributed variable $p \sim \beta(a, b)$ and a Bernoulli observation $b \sim \text{Ber}(p)$, $b \in \{\texttt{true}, \texttt{false}\}$, the posterior distribution of p given an observation of the value x is given by:

$$P(p|b = \texttt{true}) = \beta(a + 1, b)(p) \tag{15}$$
$$P(p|b = \texttt{false}) = \beta(a, b + 1)(p) \tag{16}$$

Therefore, we simply need to count the number of `true` and `false` observations for any `flip` which references p, which gives the following likelihood representation and posterior update rule:

$$\texttt{Leaf}\big(\{p\},\ (t, f)\big) \star_\beta \texttt{Leaf}\big(\{p\},\ (t', f')\big) = \texttt{Leaf}\big(\{p\},\ (t{+}t', f{+}f')\big) \tag{17}$$
$$\texttt{Leaf}\big(\{p\},\ (t, f)\big)\big[\beta(a, b)\big] = \beta(a + t, b + f) \tag{18}$$

We note explicitly that these leaf likelihoods do *not* represent a Beta-Binomial likelihood, and there is no associated factor $\binom{t+f}{t}$. This is because each observation of `true` or `false` corresponds to a *specific* identified Boolean variable. Therefore this is the likelihood of a series of Bernoulli observations $p^t(1 - p)^f$.

We also allow users to directly observe whether the value of a beta variable is equal to some real value $r \in [0, 1]$. This yields a second type of beta leaf $\texttt{Leaf}\big(\{p\},\ \texttt{eq}(r)\big)$, where the product $\star_\beta$ must add a 'correction factor' that accounts for the likelihood of observations which reference the symbolic beta variable:

$$\texttt{Leaf}\big(\{p\},\ (t, f)\big) \star_\beta \texttt{Leaf}\big(\{p\},\ \texttt{eq}(r)\big) = r^t \cdot (1 - r)^f \cdot \texttt{Leaf}\big(\{p\},\ \texttt{eq}(r)\big) \tag{19}$$
$$\texttt{Leaf}\big(\{p\},\ \texttt{eq}(r)\big)\big[\beta(a, b)\big] = \delta(r) \tag{20}$$

C.2 DIRICHLET-CATEGORICAL

Categorical variables are represented using a stick-breaking process. Since BDDs can only contain nodes which reference boolean variables, we must describe our categorical outcomes using only boolean variables. This is done using $n - 1$ boolean variables, each representing a statement 'choose category i given we did not choose $j < i$', where the statement 'category n was chosen' is given by setting all other variables to false. Equivalently, we consider each of the boolean variables x_i in turn; if x_i is `true` we choose category i, if `false` we proceed to the next. This is illustrated in figure 3 for a categorical variable with 4 outcomes (although note that this will be represented as n separate BDD guards, one for each categorical outcome).

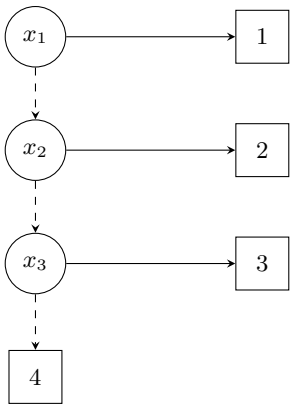


Figure 3: Stick-breaking encoding of a categorical over 4 categories using 3 boolean variables x_1, x_2, x_3. Each x_i is the i-th stick: a *hit* (true, solid) selects category i immediately, while a *miss* (false, dashed) proceeds to the next stick. Breaking all 3 sticks yields the leftover category 4.

A Dirichlet vector is represented by $n-1$ beta variables associated with the $n-1$ boolean variables in the stick-breaking process, making the Dirichlet likelihood representation similar to the beta case but with $n-1$ `true`/`false` counts, giving the following product and posterior update, with real-valued observations of the Dirichlet also handled similarly to the beta case:

$$\texttt{Leaf}\big(\{p\},\ (t_1, f_1), \ldots, (t_{n-1}, f_{n-1})\big) \star_{\text{Dir}} \texttt{Leaf}\big(\{p\},\ (t'_1, f'_1), \ldots, (t'_{n-1}, f'_{n-1})\big)$$
$$=$$
$$\texttt{Leaf}\big(\{p\},\ (t_1 + t'_1, f_1 + f'_1), \ldots, (t_{n-1} + t'_{n-1}, f_{n-1} + f'_{n-1})\big) \tag{21}$$

$$\texttt{Leaf}\big(\{p\},\ (t_1, f_1), \ldots, (t_{n-1}, f_{n-1})\big)\big[\,\text{Dir}(\alpha_1, \ldots, \alpha_{n-1}, \alpha_n)\big] = \text{Dir}(\alpha_1 + t_1, \ldots, \alpha_{n-1} + t_{n-1}, \alpha_n + f_{n-1}) \tag{22}$$

Note that the invariant $f_{i-1} = t_i + f_i$ is preserved, since the number of times we reach stick i is the same as the number of times we reach stick $i-1$ and do not select it.

Note that our implementation allows users to take an element at index i of a Dirichlet-distributed vector of probabilities and call `flip` with it. In the case where the `flip` is `true`, this is the same as observing category i in a categorical call. However, the `false` case can be thought of as 'some category that was *not* i was observed', i.e. a mixture over all other categorical observations. This can be represented as a `Sum` node over Dirichlet likelihoods of the form above.

We note that if instead of BDDs, we used more expressive multiway decision diagrams (MDDs) which allow nodes to have multiple children, we could instead represent categorical variables directly as an MDD variable and represent the Dirichlet likelihood as a vector of counts for each category. We leave it to future work to explore whether this representation is more efficient in practice.

C.3 NORMAL-NORMAL

Our implementation supports uni- and multivariate Gaussian distributions, taking affine combinations of them, and observing outcomes such as $5x + 2y - z = 1.3$ for Gaussian variables x, y, z. The likelihood representation for Gaussians is different from beta and Dirichlet variables in that it has a scope with more than one variable. It holds a set of affine observations of the form $\sum a_i g_i = v$ for $a_i, v \in \mathbb{R}$, g_i any Gaussian variable, represented as a sparse matrix A_{ij} and a vector b_j where for each row j we only store the coefficients of the non-zero indices as $\{i \mapsto a_i | a_i \neq 0\}$. This encodes the ϵ-parameterized likelihood which is an indicator function for the set $|\sum_j A_i j g_i - b_j| < \epsilon/2$

We also store a complementary set of *excluded* observations, again represented as a sparse matrix A_{ij} and a vector b_j, this time representing the indicator function for the set $|\sum_j A_i j g_i - b_j| \geq \epsilon/2$. Since expressions of the form $5x + 2y - z = 1.3$ have boolean type in our language, their negation introduces a likelihood of this form.

The product is given by the union of these observation sets, and updating the scope to correctly reflect the set of Gaussian

variables which are now correlated:

$$\begin{aligned}&\texttt{Leaf}\big(\{w,x\},\ \{x-w=0, 5x=6\}\big) \star_{\mathcal{N}} \texttt{Leaf}\big(\{x,y,z\},\ \{x+y=0, x+y-3z=1.5\}\big)\\ &\quad= \texttt{Leaf}\big(\{w,x,y,z\},\ \{x-w=0, 5x=6, x+y=0, x+y-3z=1.5\}\big)\end{aligned} \tag{23}$$

Excluded observations are combined similarly.

The posterior update equation for the joint prior over the variables $(x_1, \ldots, x_n) = \mathbf{x} \sim \mathcal{N}\mu, \mathbf{\Sigma}$, given the matrix representation $A\mathbf{x} = \mathbf{b}$ (where we assume this linear system has a valid solution, hence a posterior exists) for the observations is given by

$$\mathbf{x} \mid A\mathbf{x} = \mathbf{b} \ \sim\ \mathcal{N}\big(\boldsymbol{\mu}', \mathbf{\Sigma}'\big) \tag{24}$$

$$\boldsymbol{\mu}' = \boldsymbol{\mu} + \mathbf{\Sigma}A^\top(A\mathbf{\Sigma}A^\top)^\dagger(\mathbf{b} - A\boldsymbol{\mu}) \tag{25}$$

$$\mathbf{\Sigma}' = \mathbf{\Sigma} - \mathbf{\Sigma}A^\top(A\mathbf{\Sigma}A^\top)^\dagger A\mathbf{\Sigma} \tag{26}$$

where $M^\dagger$, the Moore-Penrose pseudoinverse, is the unique matrix such that

$$MM^\dagger M = M \tag{27}$$

$$M^\dagger MM^\dagger = M^\dagger \tag{28}$$

$$(MM^\dagger)^\top = MM^\dagger \tag{29}$$

$$(M^\dagger M)^\top = M^\dagger M \tag{30}$$

Note that the exclusion set does not enter into the posterior calculation (unless it conflicts with the observations, in which case the posterior is undefined). It is only used to calculate the appropriate marginal likelihood Z.

C.4 GAMMA-GAMMA-POISSON

We support two forms of likelihoods for variables γ with gamma priors $\Gamma(\alpha, \beta)$ - Poisson likelihoods and gamma ratios. Both likelihoods are represented within a shared form of `Leaf` node, as one γ variable may be observed in many different ways which must be tracked by a single likelihood.

The Poisson likelihood is represented by an interval $[a, b],\ a \in \mathbb{Z}_{\geq 0}\ b \in \mathbb{Z}_{\geq 0} \cup \{\infty\}$. The likelihood represented by $[a, b]$ is $f_\epsilon(\lambda \mid a \leq k \leq b) = \frac{\Gamma(a,\lambda)}{\Gamma(a)} - \frac{\Gamma(b+1,\lambda)}{\Gamma(b+1)}$ for a Poisson random variable $k \sim \text{Poisson}(\lambda)$, where $Q(a, \lambda) = \Gamma(a, \lambda)/\Gamma(a)$ is the regularized upper incomplete gamma function and the second term is 0 if $b = \infty$. For a `Leaf` node containing only Poisson observations referencing Poisson variables k_i and a shared gamma rate λ, we have the following product:

$$\texttt{Leaf}\big(\lambda,\ \{k_i \mapsto [a_i, b_i]\}\big) \star \texttt{Leaf}\big(\lambda,\ \{k_j \mapsto [c_j, d_j]\}\big) = \texttt{Leaf}\big(\lambda,\ \{k_i \mapsto [a_i, b_i] \cap [c_i, d_i]\}\big) \tag{31}$$

where for any Poisson variables k_i not appearing in one of the `Leaf` nodes we can implicitly set $[a_i, b_i] = [0, \infty]$.

The gamma ratio likelihood is represented by a weighted directed graph G, where the nodes i represent different gamma variables λ_i, and the edges $w(i, j) = r$ represent events $\mid \frac{\lambda_i}{\lambda_j} - r \mid < \epsilon/2$. This means the overall graph represents a likelihood $f_\epsilon(\lambda_1, \ldots \lambda_n \mid \forall w(i, j) = r\,, \mid \frac{\lambda_i}{\lambda_j} - r \mid < \epsilon/2) = \mathbb{1}_S$ where S is the intersection $\bigcap_{(i,j)\in E(G)} \left(\mid \frac{\lambda_i}{\lambda_j} - r \mid < \epsilon/2 \right)$. The product in this case is given by a graph union, where the vertices in the graph use the usual set union (since the vertex i always refers to the same gamma variable) and the edges take the *disjoint* union (since the observations $\mid \frac{\lambda_i}{\lambda_j} - r \mid < \epsilon/2$, $\mid \frac{\lambda_i}{\lambda_j} - s \mid < \epsilon/2$ are distinct):

$$\texttt{Leaf}\big(\{\lambda_i\}_{i\in S},\ G_1(V_1, E_1, w_1)\big) \star \texttt{Leaf}\big(\{\lambda_i\}_{i\in T},\ G_2(V_2, E_2, w_2)\big) = \texttt{Leaf}\big(\{\lambda_i\}_{i\in S\cup T},\ G(V_1 \cup V_2, E_1 \sqcup E_2, w_1 \sqcup w_2)\big) \tag{32}$$

The full gamma `Leaf` node combines both of these likelihoods, with the product being performed componentwise on the Poisson and ratio components. These likelihoods both have tractable posteriors for a gamma prior, although note that Poisson inequality observations (as opposed to equality) will give a gamma *mixture* posterior, which is still representable within our framework.

D WEIGHTED MODEL COUNTING ALGORITHM

Given a BDD with N nodes and an associated weight map γ giving weights in a semiring $(R, \oplus, \otimes, \bar{0}, \bar{1})$, weighted model counting can be performed in $O(N)$ time using the algorithm 2.

Algorithm 2 Weighted Model Count over a semiring

Require: BDD node n; semiring $(R, \oplus, \otimes, \bar{0}, \bar{1})$; weight function w mapping each literal to an element of R
Ensure: The weighted model count of n, an element of R

function WMC(n)
  if $n = \bot$ **then return** $\bar{0}$
  end if ▷ false leaf
  if $n = \top$ **then return** $\bar{1}$
  end if ▷ true leaf
  if $n \in$ *cache* **then return** *cache*$[n]$
  end if
  $x \leftarrow \textsc{Var}(n)$ ▷ variable tested at n
  $n_h \leftarrow \textsc{High}(n), \quad n_\ell \leftarrow \textsc{Low}(n)$
  $h \leftarrow w(x) \otimes \text{WMC}(n_h)$ ▷ branch $x =$ true
  $l \leftarrow w(\neg x) \otimes \text{WMC}(n_\ell)$ ▷ branch $x =$ false
  cache$[n] \leftarrow h \oplus l$
  return *cache*$[n]$
end function

E BENCHMARK PROGRAMS

This appendix lists the full source of each benchmark program used in our evaluation. Programs are written in Hybrid Pluck's surface syntax: `(query name ...)` declares an inference problem, `(Posterior q evidence)` returns the posterior over *q* given the *evidence* conjunction, and `(Marginal e)` returns the unconditioned marginal of *e*. Continuous observations are expressed with `real_eq` (condition a continuous draw on exact equality to an observed value) and `prob_eq`; discrete observations use `iff` / `native_eq`. Several programs are ports of the Alea.jl / Dice benchmark suite; where the original relied on truncated or bit-blasted distributions, Hybrid Pluck uses the corresponding exact distribution, which we note inline.

E.1 COINBIAS

A Beta–Bernoulli model: a coin bias $a \sim \text{Beta}(2, 5)$ is observed through five flips with outcomes $[1, 1, 0, 1, 0]$. The query returns the full posterior over a, which by conjugacy is $\text{Beta}(5, 7)$. (Dice reports only the expectation $\mathbb{E}[a]$.)

```
(define (obs-flip p y) (iff (flip p) y))
(define (obs-all p ys)
  (match ys
    Nil => (True)
    Cons y rest => (and (obs-flip p y) (obs-all p rest))))

(query coinBias
  (let ((a (beta 2.0 5.0)))
    (Posterior a (obs-all a [true, true, false, true, false]))))
```

E.2 WEEKEND

$isWeekend \sim \text{Bernoulli}(2/7)$. The hour someone wakes is $\mathcal{N}(5, 4)$ on a weekend and $\mathcal{N}(2, 4)$ on a weekday. We observe an hour of 6.0 and return the posterior over $isWeekend$. (Dice truncates the hour to $[0, 8]$; Hybrid Pluck uses the exact Gaussian.)

```
(query weekend
  (let ((isWeekend (flip 0.2857142857142857)))
    (Posterior isWeekend
      (real_eq (if isWeekend (normal 5.0 4.0) (normal 2.0 4.0)) 6.0))))
```

E.3 SPACEX

A rocket's completion time is the sum of three independent stage times: *engines* $\sim \mathcal{N}(5,1)$, *first_stage* $\sim \mathcal{N}(10,3)$, and *second_stage* $\sim \mathcal{N}(15,3)$. The query returns the full marginal of the total, $\mathcal{N}(30, \sqrt{19})$. (Dice truncates each stage to $[0, 64]$ and reports the expectation; Hybrid Pluck uses exact Gaussians.)

```
(query spacex
  (Marginal
    (let ((engines (normal 5.0 1.0))
          (first_stage (normal 10.0 3.0))
          (second_stage (normal 15.0 3.0)))
      (+. (+. engines first_stage) second_stage))))
```

E.4 CONJUGATE_GAUSSIANS

A conjugate Gaussian model with prior $a \sim \mathcal{N}(0,1)$. Two data points, 8.0 and 9.0, are each observed as a noisy reading $a + \mathcal{N}(0,1)$. The query returns the posterior over a, which is $\mathcal{N}(17/3,\ 1/\sqrt{3}) \approx \mathcal{N}(5.667, 0.577)$. (Dice uses a bit-blasted normal truncated to $[-16, 16]$; Hybrid Pluck uses the exact, untruncated conjugate update.)

```
(define (observe-noisy a y) (real_eq (normal a 1.0) y))

(query conjugate_gaussians
  (let ((a (gaussian 0.0 1.0)))
    (Posterior a
      (and (observe-noisy a 8.0)
           (observe-noisy a 9.0)))))
```

E.5 GPA

Two grading systems. School 1 (chosen with prob. 0.25) reports a GPA of $4 \cdot \mathrm{Beta}(8,2)$ with prob. 0.95, else a point mass at 4.0 or 0.0; School 2 reports $8 \cdot \mathrm{Beta}(5,5)$ with prob. 0.99, else a point mass at 8.0 or 0.0. We observe a GPA of 0.0 and ask which school the student is from. Because Hybrid Pluck cannot represent a *scaled* Beta directly, the scale is folded into the observation instead (scale $\cdot\ \mathrm{Beta}(a,b) = v \iff \mathrm{Beta}(a,b) = v/\mathrm{scale}$), which is exact at the observed boundary value $v = 0$ where the Beta density vanishes.

```
;; scale * Beta(a,b) == v   <=>   Beta(a,b) == v/scale
(define (scaled-beta-eq a b scale v)
  (prob_eq (beta a b) (/. v scale)))

;; Evidence that school 1's GPA equals v.
(define (gpa1-eq v)
  (if (flip 0.95)
      (scaled-beta-eq 8.0 2.0 4.0 v)   ; continuous 4*Beta(8,2)
      (if (flip 0.15)
          (real_eq 4.0 v)              ; point mass at 4.0
          (real_eq 0.0 v))))           ; point mass at 0.0

;; Evidence that school 2's GPA equals v.
(define (gpa2-eq v)
  (if (flip 0.99)
      (scaled-beta-eq 5.0 5.0 8.0 v)   ; continuous 8*Beta(5,5)
      (if (flip 0.1)
```

```
          (real_eq 8.0 v)              ; point mass at 8.0
          (real_eq 0.0 v))))           ; point mass at 0.0

(define observed-gpa 0.0)

(query GPA
  (let ((n (flip 0.25)))
    (Posterior n
      (if n (gpa1-eq observed-gpa) (gpa2-eq observed-gpa)))))
```

E.6 CLICKGRAPH

A click-graph similarity model. A shared *similarityAll* $\sim$ Uniform$[0, 1]$ governs five trials. In each trial we draw *sim* $\sim$ Bernoulli(*similarityAll*) and a click probability $b_1 \sim$ Uniform$[0, 1]$; the second probability b_2 equals b_1 if the items are similar, else is a fresh Uniform$[0, 1]$. The two observed clicks are Bernoulli(b_1) and Bernoulli(b_2). We condition on the click data and return the posterior over *similarityAll*. (Uniform$[0, 1]$ is expressed as Beta$(1, 1)$.)

```
(define (trial sa c0 c1)
  (let ((sim (flip sa))
        (b1 (beta 1.0 1.0))
        (b2 (if sim b1 (beta 1.0 1.0))))
    (and (iff (flip b1) c0)
         (iff (flip b2) c1))))

(define (all-trials sa c0s c1s)
  (match c0s
    Nil => (True)
    Cons c0 rest0 =>
      (match c1s
        Cons c1 rest1 =>
          (and (trial sa c0 c1) (all-trials sa rest0 rest1)))))

(query clickGraph
  (let ((similarityAll (beta 1.0 1.0)))
    (Posterior similarityAll
      (all-trials similarityAll
        [true, true, true, false, false]
        [true, true, true, false, false]))))
```

E.7 CLINICALTRIAL1

isEffective $\sim$ Bernoulli(0.5). If the treatment is effective, the control and treated groups have independent success probabilities; otherwise they share a single probability. All priors are Uniform$[0, 1]$ = Beta$(1, 1)$. We observe 100 control and 100 treated outcomes and return the posterior over *isEffective*.

```
(define (obs-flip p y) (iff (flip p) y))
(define (obs-all p ys)
  (match ys
    Nil => (True)
    Cons y rest => (and (obs-flip p y) (obs-all p rest))))

(define control [false, false, true, true, true, false, true, true, false, false, false,
    false, false, false, true, false, true, false, false, true, true, true, true, true,
    true, false, false, true, true, true, false, false, true, true, true, false, false,
    true, true, false, false, true, false, false, true, false, false, true, true, true,
    false, false, true, false, true, false, true, true, false, true, false, false, true,
    true, true, false, false, false, true, false, true, true, false, false, true, true,
    true, false, false, false, true, true, false, false, false, true, true, false, false,
    false, true, false, true, true, false, false, false, true, false, true])
```

```
(define treated [false, true, false, false, true, true, true, true, true, true, false,
    false, true, true, false, true, false, false, true, false, true, false, true, false,
    false, false, false, true, true, false, false, false, true, true, true, false, true,
    true, true, true, true, true, true, false, true, false, true, false, false, true
    , false, true, true, true, true, true, true, false, false, true, false, true, false,
    false, false, true, false, true, false, true, false, false, true, true, false, true,
    false, false, false, true, false, false, true, true, true, true, false, false, true,
    true, false, true, true, false, false, true, false, false, false])

(query clinicalTrial1
  (let ((isEffective (flip 0.5))
        (probControl (beta 1.0 1.0))
        (probTreated (beta 1.0 1.0))
        (probAll (beta 1.0 1.0)))
    (Posterior isEffective
      (if isEffective
        (and (obs-all probControl control)
             (obs-all probTreated treated))
        (and (obs-all probAll control)
             (obs-all probAll treated))))))
```

E.8 CLINICALTRIAL2

isEffective $\sim$ Bernoulli(0.5). The treated group's success probability is *probIfTreated* $\sim$ Uniform$[0, 1]$; the control group's is a separate Uniform$[0, 1]$ if the treatment is effective, else it shares *probIfTreated*. We observe five control and five treated outcomes, condition on *isEffective*, and return the posterior over *probIfControl*.

```
(define (obs-flip p y) (iff (flip p) y))
(define (obs-all p ys)
  (match ys
    Nil => (True)
    Cons y rest => (and (obs-flip p y) (obs-all p rest))))

(define control [false, false, true, false, false])
(define treated [true, false, true, true, true])

(query clinicalTrial2
  (let ((isEffective (flip 0.5))
        (probIfTreated (beta 1.0 1.0))
        (probIfControl (if isEffective (beta 1.0 1.0) probIfTreated)))
    (Posterior probIfControl
      (and isEffective
        (and (obs-all probIfControl control)
             (obs-all probIfTreated treated))))))
```

E.9 NORMAL_MIXTURE

A two-component Gaussian mixture with shared component means. The mixture weight is $\theta \sim$ Uniform$[0, 1]$, and the means $\mu_1 \sim \mathcal{N}(-10, 1)$ and $\mu_2 \sim \mathcal{N}(10, 1)$ are drawn once and shared across all points. Each datapoint is $\mu_1 + \mathcal{N}(0, 1)$ with prob. θ, else $\mu_2 + \mathcal{N}(0, 1)$, observed equal to y. The query returns the posterior over θ. Because the means are shared while each point's component assignment is a latent flip(θ), exact inference must enumerate the coupled assignments, which grows exponentially in the number of points; this port therefore uses the first 16 of the benchmark's 40 datapoints.

```
(define (obs-point theta mu1 mu2 y)
  (real_eq
    (if (flip theta) (+. mu1 (gaussian 0.0 1.0)) (+. mu2 (gaussian 0.0 1.0)))
    y))
```

```
(define (obs-all theta mu1 mu2 ys)
  (match ys
    Nil => (True)
    Cons y rest => (and (obs-point theta mu1 mu2 y) (obs-all theta mu1 mu2 rest))))

;; first 16 of the 40 benchmark datapoints (exact inference scales exponentially in N)
(define ys [10.0787617156523, -9.51866467444093, 9.73922587306449, 11.5662681883816,
    9.62798489000074, -9.60265090119919, 8.90114345923455, 10.866328444034,
    10.5347883361026, 8.60577222463504, -10.625227428575, -9.08615131315038,
    -8.99280046532217, 9.43533905966944, 8.92696803962787, 9.642301288006])

(query normal_mixture
  (let ((theta (beta 1.0 1.0))
        (mu1 (normal -10.0 1.0))
        (mu2 (normal 10.0 1.0)))
    (Posterior theta (obs-all theta mu1 mu2 ys))))
```

E.10 RUN_WALK_HRM

A heart-rate hidden Markov model with learned transition probabilities. The shared latent parameters are a resting rate *base* $\sim \mathcal{N}(70, 20)$, a running increment *incr* $\sim \mathcal{N}(50, 5)$, and transition probabilities *p_run*, *p_walk* $\sim \mathrm{Beta}(1, 1)$. The hidden `running` state starts as a fair flip; at each step the reading is $\mathcal{N}(\textit{base} + \textit{incr}, 5)$ when running and $\mathcal{N}(\textit{base}, 5)$ otherwise, after which the state transitions. The query returns the joint posterior over the final running state and *base*.

```
(define obs [82.0, 81.0, 80.0, 83.0, 120.0, 124.0, 120.0, 105.0])

(define (gen-params)
  (let
   ((start-rate (beta 1.0 1.0))
    (stop-rate (beta 1.0 1.0))
    (base-hr (normal 70.0 20.0))
    (incr (normal 40.0 20.0)))
    (Pair (Pair start-rate stop-rate)
          (Pair base-hr (+. base-hr incr)))))

(define (reverse-helper xs ys)
  (match xs
    Nil => ys
    Cons x xs => (reverse-helper xs (Cons x ys))))
(define (reverse xs)
  (reverse-helper xs []))

(define (hmm params state past hrs t)
  (match t
    0 => (Pair (reverse hrs) (reverse past))
    S t => (let
            ((running state)
             (hr (normal (if running
                             (snd (snd params))
                             (fst (snd params))) 5.0))
             (new-running (if running
                              (not (flip (snd (fst params))))
                              (flip (fst (fst params))))))
             (hmm params
                  new-running
                  (Cons state past)
                  (Cons hr hrs)
                  t)))))
```

```
    (query
     (let ((params (gen-params))
           (res (hmm params (flip 0.5) [] [] (length obs)))
           (hrs (fst res))
           (sts (snd res)))
       (Posterior (Pair (last sts) (fst (snd params))) (list=? real=? hrs obs))))
```

E.11 SOCCER

A soccer tournament with latent team identities and gamma–Poisson goal counts. Each team's identity is drawn from a shared Categorical with a Dirichlet$(1,1,1)$ prior over $\{\text{defend}, \text{balanced}, \text{attack}\}$, making this a Dirichlet–categorical conjugate model. Each team independently draws six Gamma$(4, 4/\text{mean})$ goal-rate priors, one per matchup index, with prior means 1.5^s. A match adds the two teams' identities; the home team (listed first) receives a $+1$ index boost and scores Poisson at the boosted index, the away team at the base index. We observe every match score and query the posterior over each team's identity.

```
(define team-names ["Brazil" "Norway" "Curacao" "Mexico" "Japan" "Senegal"])

(define-type game (Game any any any any))   ;; team-i  team-j  goals-i  goals-j

(define scores
  ;; full double round robin (all 30 ordered pairs); each team hosts 5, visits 5.
  ;; home team listed first (gets the +1 boost); scores sampled from the model.
  [(Game 0 1 @2 @4) (Game 0 2 @2 @2) (Game 0 3 @2 @2) (Game 0 4 @2 @2) (Game 0 5 @3 @0)
     ;; Brazil  (H)
   (Game 1 0 @3 @4) (Game 1 2 @2 @1) (Game 1 3 @3 @2) (Game 1 4 @2 @0) (Game 1 5 @0 @2)
     ;; Norway  (H)
   (Game 2 0 @2 @1) (Game 2 1 @4 @1) (Game 2 3 @1 @1) (Game 2 4 @0 @0) (Game 2 5 @0 @0)
     ;; Curacao (H)
   (Game 3 0 @1 @3) (Game 3 1 @6 @2) (Game 3 2 @2 @1) (Game 3 4 @0 @1) (Game 3 5 @2 @1)
     ;; Mexico  (H)
   (Game 4 0 @1 @0) (Game 4 1 @4 @0) (Game 4 2 @1 @0) (Game 4 3 @1 @2) (Game 4 5 @1 @1)
     ;; Japan   (H)
   (Game 5 0 @0 @2) (Game 5 1 @3 @2) (Game 5 2 @0 @0) (Game 5 3 @0 @1) (Game 5 4 @0 @0)])
     ;; Senegal (H)

;; --- identity ---------------------------------------------------------------
(define (draw-identity theta)
  ;; categorical returns a Native(Int) in {@0,@1,@2}; map it to a Peano nat.
  (let ((c (categorical theta)))
    (if (native_eq c @0) 0 (if (native_eq c @1) 1 2))))

(define (identity-label id)
  (if (== id 0) "defend" (if (== id 1) "balanced" "attack")))

;; --- rates ------------------------------------------------------------------
;; shape k = 4 concentrates each prior around its mean (CV = 1/sqrt(k) = 0.5).
(define rate-shape 4.0)
;; a rate prior with a given prior MEAN: Gamma(k, k/mean) has mean exactly `mean`.
(define (rate-prior mean) (gamma rate-shape (/. rate-shape mean)))
;; position p = 0..5 -> identity-sum s = p-2 -> prior mean = 1.5^s.
(define (make-rates)
  [(rate-prior (/. 4.0 9.0))   ;; s=-2  ~ 0.444
   (rate-prior (/. 2.0 3.0))   ;; s=-1  ~ 0.667
   (rate-prior 1.0)            ;; s= 0
   (rate-prior 1.5)            ;; s= 1
   (rate-prior 2.25)           ;; s= 2
   (rate-prior 3.375)])        ;; s= 3  (home-boosted top)
```

```
;; --- teams ------------------------------------------------------------------
;; a team bundles its identity (Peano nat) with its rate priors.
(define (make-team theta) (Pair (draw-identity theta) (make-rates)))
(define (team-id t)    (fst t))
(define (team-rates t) (snd t))

;; draw `n` IID teams (n a Peano nat); they share the identity prior theta.
(define (make-teams theta n)
  (match n
    O => (Nil)
    S m => (Cons (make-team theta) (make-teams theta m))))

;; --- observation ------------------------------------------------------------
;; observe one match. `home` gets the +1 home boost (index p+1); `away` is at p.
(define (observe-match home away g-home g-away)
  (let ((p (+ (team-id home) (team-id away))))        ;; base matchup index in {0..4}
    (and (native_eq (poisson (index (+ p 1) (team-rates home))) g-home)  ;; home: p+1
         (native_eq (poisson (index p (team-rates away))) g-away))))     ;; away: p

;; fold the score list into one evidence conjunction.
(define (observe-scores teams gs)
  (match gs
    Nil => (True)
    Cons g rest =>
      (match g
        Game i j gi gj =>
          (and (observe-match (index i teams) (index j teams) gi gj)
               (observe-scores teams rest)))))

;; --- query ------------------------------------------------------------------
;; Marginal posterior over one team's identity given every match score. This
;; sums over the other teams' identities (a full 3^6 enumeration).
(define (identity-posterior i)
  (let ((theta (dirichlet 1.0 1.0 1.0)))
    (let ((teams (make-teams theta (length team-names))))
      (Posterior (identity-label (team-id (index i teams)))
                 (observe-scores teams scores)))))

(query Brazil  (identity-posterior 0))
(query Norway  (identity-posterior 1))
(query Curacao (identity-posterior 2))
(query Mexico  (identity-posterior 3))
(query Japan   (identity-posterior 4))
(query Senegal (identity-posterior 5))
```

E.12 POLYREG

Polynomial regression with outlier detection. A random polynomial is generated with degree drawn from a truncated geometric distribution (parameter 0.3, maximum degree 5) and coefficients drawn independently from $\mathcal{N}(0, 100)$. The outlier model `One` marks a single uniformly chosen point as an outlier (emitted from $\mathcal{N}(0, 200)$ rather than from the curve plus $\mathcal{N}(0, 1)$ noise). Conditioning the generated data on the observed *ys*, the query returns the posterior over which indices are outliers.

```
(define xs [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0])
(define ys [1.3998703068665284, 6.235613006596935, 14.930938208255952, 27.76621455275837,
   44.63428250334452, 65.39877806912918, 90.16345175668341, 118.94703945597995,
   151.7909055407857, 188.5411680368732])

;; Generate a random polynomial: degree from a truncated Geometric (parameter p,
```

```
;; maximum degree max-degree); each coefficient drawn from N(0, 100).
(define (random-polynomial p max-degree)
  (match max-degree
    O => []
    S n => (if (flip p)
               (Nil)
               (Cons (normal 0.0 100.0)
                     (random-polynomial p n)))))

(define (evaluate-polynomial coeffs x)
  (match coeffs
    Nil => 0.0
    Cons c cs => (+. c (*. x (evaluate-polynomial cs x)))))

(define-type outlier-model (None) (One) (All))

(define (generate-outliers model)
  (match model
    None => (map (fn _ -> false) xs)
    One => (let ((i (uniform 0 1 2 3 4 5 6 7 8 9)))
             (map (fn j -> (nat=? i j)) (range (length xs))))
    All => (let ((p (beta 1.0 1.0))) (map (fn _ -> (flip p)) xs))))

(define (describe-degree n)
  (index n ["Zero", "Constant", "Linear", "Quadratic", "Cubic", "Quartic", "Quintic"]))

(define-type sample (Trace curve outliers data))

(define (zip xs ys)
  (match xs
    Nil => []
    Cons x xs => (Cons (Pair x (car ys))
                       (zip xs (cdr ys)))))

(define (model outlier-model)
  (let
    ((curve (random-polynomial 0.3 5))
     (outliers (generate-outliers outlier-model))
     (data (map (fn xo -> (if (snd xo)
                              (normal 0.0 200.0)
                              (normal (evaluate-polynomial curve (fst xo)) 1.0)))
                (zip xs outliers))))
    (Trace curve outliers data)))

(define (true-indices xs)
  (match xs
    Nil => (Nil)
    Cons x xs =>
      (let ((rest (map (fn x -> (+ x 1)) (true-indices xs))))
        (if x (Cons 0 rest) rest))))

(query infer-params
  (match (model (One))
    Trace curve outliers data =>
      (Posterior (true-indices outliers) (list=? real=? data ys))))
```