Probabilistic Inference for Datalog with Correlated Inputs

JINGBO WANG, Purdue University, USA
SHASHIN HALALINGAIAH, University of Texas at Austin, USA
WEIYI CHEN, Purdue University, USA
CHAO WANG, University of Southern California, USA
IŞIL DILLIG, University of Texas at Austin, USA

Probabilistic extensions of logic programming languages, such as ProbLog, integrate logical reasoning with probabilistic inference to evaluate probabilities of output relations; however, prior work does not account for potential statistical correlations among input facts. This paper introduces Praline, a new extension to Datalog designed for precise probabilistic inference in the presence of (partially known) input correlations. We formulate the inference task as a constrained optimization problem, where the solution yields sound and precise probability bounds for output facts. However, due to the complexity of the resulting optimization problem, this approach alone often does not scale to large programs. To address scalability, we propose a more efficient δ-exact inference algorithm that leverages constraint solving, static analysis, and iterative refinement. Our empirical evaluation on challenging real-world benchmarks, including side-channel analysis, demonstrates that our method not only scales effectively but also delivers tight probability bounds.

CCS Concepts: • **Mathematics of computing → Probabilistic inference problems**; **Mathematical optimization**; • **Theory of computation → Constraint and logic programming**.

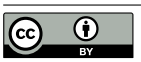
Additional Key Words and Phrases: probabilistic logic programming, constrained optimization, type inference

ACM Reference Format:
Jingbo Wang, Shashin Halalingaiah, Weiyi Chen, Chao Wang, and Işıl Dillig. 2025. Probabilistic Inference for Datalog with Correlated Inputs. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 280 (October 2025), 28 pages. https://doi.org/10.1145/3763058

1 Introduction

Logic programming languages are powerful tools for modeling and reasoning about complex systems using well-defined rules, and they are also widely used in graph analysis, bioinformatics, and program analysis tasks [44, 63] such as detecting race conditions [40, 62] and side channels [57, 58]. However, real-world scenarios often involve uncertainty and incomplete information that traditional logic programming cannot handle effectively. Probabilistic extensions to logic programming languages, such as ProbLog [16, 17] and PPDL [5], enhance traditional logic programming by incorporating probabilistic reasoning. For example, in medical diagnosis, symptom-disease associations are inherently probabilistic, and quantifying disease likelihood based on symptoms can enhance diagnostic accuracy. Such probabilistic reasoning is also useful for applications of logic programming in program analysis tasks [43]: in the context of side channel detection, understanding the probability of information leakage can significantly enhance security assessments.

Authors' Contact Information: Jingbo Wang, Purdue University, West Lafayette, USA, wang6203@purdue.edu; Shashin Halalingaiah, University of Texas at Austin, Austin, USA, shashin@cs.utexas.edu; Weiyi Chen, Purdue University, West Lafayette, USA, chen5332@purdue.edu; Chao Wang, University of Southern California, Los Angeles, USA, wang626@usc.edu; Işıl Dillig, University of Texas at Austin, Austin, USA, isil@cs.utexas.edu.



ACM 2475-1421/2025/10-ART280
https://doi.org/10.1145/3763058

However, these probabilistic extensions to logic programming languages typically assume the independence of input facts to simplify inference [5, 16, 37]. In many real-world scenarios, this assumption does not hold: for example, in medical diagnostics, certain symptoms are often correlated, and treating them as independent can lead to incorrect diagnoses. Similarly, in side-channel analysis, input facts such as memory accesses and branch outcomes are frequently correlated, and ignoring these correlations can produce unsound assessments.

This work aims to develop a novel probabilistic Datalog framework that can accommodate arbitrary statistical correlations between inputs. However, the removal of the independence assumption introduces several challenges: First, it significantly increases the complexity of probabilistic inference, requiring consideration of joint probabilities of inputs rather than treating them in isolation. Second, since the exact conditional dependencies between input facts may not be known a priori, there can be *implicit* dependencies that are not explicitly specified by users. Finally, due to *partial* knowledge about input dependencies, it may not always be possible to compute the *exact* probability of an output fact, necessitating the computation of lower and upper probability bounds. Our proposed framework addresses these challenges, providing an effective approach for accurate and reliable probabilistic reasoning in complex scenarios that involve correlated inputs.

At the core of our approach is a formulation of probabilistic inference as a constrained optimization problem over *joint probability variables*, which capture the joint probability distribution of correlated input facts. Probabilities specified in the Datalog program are translated into a system of constraints, and the probability of each output relation is expressed symbolically in terms of these variables. Our method then solves a constrained optimization problem to compute upper and lower bounds on the probability of the desired output fact. However, a key challenge with this approach is that the generated constrained optimization problems can be very challenging to solve, particularly due to the non-linear nature of the objective function. To address this difficulty, we also present a more scalable δ-exact algorithm [22] that uses the optimization approach in a targeted way to solve simpler sub-problems. Our proposed method first computes *approximate* probability bounds and then iteratively refines them until they are within a user-specified distance δ of the true probability bounds. As illustrated in Figure 1, our technique comprises of three steps. First, it performs a combination of static analysis and constraint solving to infer whether a pair of facts are positively or negatively correlated, or independent. In the second step, it leverages the results of this "correlation type analysis" to compute approximate bounds on the probabilities of each predicate. Finally, it iteratively refines these approximate bounds to a user-specified tightness δ — i.e., upon termination, the computed lower and upper bounds on the probabilities are guaranteed to be within δ of the ground truth.

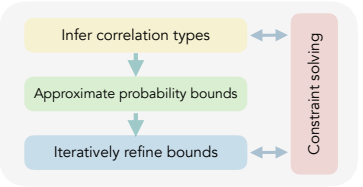


Fig. 1. Overview of our approach

We have implemented the proposed idea in a new Datalog variant called Praline[1] and evaluate it on two domains, namely side channel vulnerability detection and inference for discrete Bayesian networks. Our experiments show that Praline can successfully infer accurate probability bounds for output facts, including for large benchmarks with hundreds of thousands of facts. We also conduct ablation studies to evaluate the impact of the three key ideas illustrated in Figure 1 and show that they have a significant impact on the scalability and precision of our approach.

To summarize, this paper makes the following key contributions:

- We introduce the first probabilistic Datalog framework that allows the user to specify arbitrary statistical dependencies between input facts.

[1]Stands for PRobAbilistic Logical InfereNce Engine

```
1  % Input facts.
2  % Known input facts about patient
3  1.00 :: chest_pain.
4  % Unknown input facts about patient
5  % Probabilities estimated from average population.
6  0.03 :: abnormal_ecg.
7  0.02 :: troponin_high.
8  ...
9  % Input correlations based on the literature
10 0.90 :: abnormal_ecg | chest_pain, troponin_high.
11 ...
12 % Inference rules for heart disease assessment
13 0.85 :: heart_disease :- abnormal_ecg, troponin_high,
       chest_pain.
14 0.60 :: heart_disease :- abnormal_ecg, high_bnp,
       chest_pain, \+troponin_high.
15 % Query
16 query(heart_disease).
```

```
1  % Input Facts about input program from Fig. 3
2  1.0 :: rand(r3).  % r3 is a random variable
3  % XOR operation highlighted in Fig. 3
4  1.0 :: xor(n5, n7, n8).
5  ...
6  0.7 :: share(n5, n8). % register sharing
7  0.8 :: dep(n5, r3). % data dependency
8  0.8 :: dep(n8, r3).
9  ...
10 % Input correlations
11 0.9 :: share(n5,n8) | xor(n5,n7,n8), dep(n5,r3).
12 ...
13 % Rules for inferring leaks
14 0.7 :: t(V1,V2) :- dep(V1, R),rand(R),dep(V2, R).
15 0.8 :: leak(V1, V2) :- share(V1, V2), t(V1,V2).
16 ...
17 % Query
18 query(leak(_,_)).
```

Fig. 2. Two simple Praline programs. Statistical correlations between inputs (in *green*) are specified using the syntax $p :: I \mid I_1, I_2$, indicating that probability of I given I_1, I_2 is p, and `\+I` indicates the negation of `I`.

- We propose a constrained optimization approach to compute exact probability bounds for outputs.
- We present a δ-exact algorithm that combines our basic constrained optimization formulation with static analysis, approximation, and iterative refinement to improve scalability.
- We perform an empirical evaluation on 30 real-world probabilistic Datalog programs consisting of large-scale program analysis tasks and Bayesian inference benchmarks. Our results show that Praline can produce precise probability bounds, while scaling to large benchmarks.

2 Motivating Examples

In this section, we provide two simple examples to motivate the probabilistic inference capabilities of Praline, highlighting scenarios where input facts are correlated but the full joint probability distribution is unknown.

Medical diagnosis. The left side of Figure 2 illustrates how a Praline program can perform probabilistic inference for medical diagnosis. Specifically, it models the probability of a patient having heart disease based on observed symptoms (e.g., chest pain) and test results (e.g., ECG, blood biomarkers). Lines 12–14 of the Praline program encode how heart disease can be inferred from various diagnostic factors based on well-established medical literature. These factors include an abnormal ECG (`abnormal_ecg`), elevated cardiac biomarkers such as troponin and B-type Natriuretic Peptide (BNP) (`troponin_high`, `high_bnp`), and the presence of chest pain (`chest_pain`).

For example, the rule at line 13 states that the presence of `heart_disease` (an output fact) can be deduced based on the input facts `abnormal_ecg` (test result), `troponin_high` (blood work result) and `chest_pain` (symptom). The input fact in line 3 represents the patient's known symptom (chest pain, with probability 1.0), while lines 6-7 encode test results that are currently unknown because the patient has not yet undergone an ECG or blood work. In clinical settings, when test results are unavailable, probabilities for these factors (e.g., `abnormal_ecg`) are estimated based on population statistics. However, chest pain is often correlated with other markers of heart disease, such as abnormal ECG findings and elevated troponin levels. This dependence is captured in line 10, where the probability of an abnormal ECG increases to 0.9 given chest pain and high troponin levels. However, the full joint probability distribution of these input facts (e.g., symptoms and test results) is typically unknown, as it depends on numerous latent factors, such as medical history or environmental exposures. The probabilistic reasoning capabilities of Praline enable accurate estimation of heart disease risk even under such *partially-known* information. For example,

assuming independence of input facts would lead us to estimate the probability of heart disease given chest pain as 0.07% whereas the true probability is in the range 1.5 − 2.1%.[2]

Quantitative program analysis. Another motivating scenario for Praline is quantitative Datalog-based program analysis, which has emerged as a powerful approach for static reasoning about program behavior [40, 43, 57, 58, 62]. In this example, we consider a (drastically) simplified version of the Datalog-based side channel detection method described in [58]. The Praline program shown on the right side of Figure 2 encodes an analysis for detecting information leaks caused by power side channels. In particular, the rules in lines 14-15 define how leakage is inferred based on data dependencies in the input program (`dep(X,Y)`), randomness (`rand(X)`), and register sharing between variables (`share(X,Y)`).

```
1 bool mChi(bool i1 ...i3, bool
2               r1 ...r3) {
3 bool b1, b2, b3... ;
4 b1 = i1 ⊕ r1; b2 = i2 ⊕ r2;
5 b3 = i3 ⊕ r3; n8 = r3 ∧ r2;
6 n7 = r3 ∨ b2;
7 n5 = n7 ⊕ n8;
8 ...}
```

Fig. 3. Masked χ function from MAC-Keccak [18]

The input facts in lines 1–5 of Figure 2 (right) correspond to properties of the analyzed program (see Figure 3). Some of these facts, such as those in lines 2 and 4, have probability 1.0 because they correspond to known characteristics of the program. For instance, `r3` is annotated by the user to be a random variable, and the `xor` operation in line 4 corresponds to a specific highlighted statement in the source program from Figure 3. However, not all input facts are deterministic. Whether two variables share a register, for example, depends on hardware constraints and register allocation policies, which introduce uncertainty. This uncertainty is reflected in line 6, where the probability of register sharing is estimated using empirical data from profiling a code corpus. Similarly, the input facts in lines 7-8 are probabilistic because they result from a pre-analysis [58, 61] that infers semantic data dependencies from syntactic ones. The probability associated with such dependencies is derived from prior empirical studies that measure how often syntactic dependencies lead to actual data dependencies in compiled programs [58].

Since register allocation and data dependencies are influenced by compiler optimizations and architectural constraints, certain input facts are naturally correlated. For example, as shown in line 11, the likelihood of two variables sharing a register is not independent of how frequently the compiler assigns dependent variables (within the same instruction) to the same register across different executions [3]. However, as in the medical diagnosis example, assuming access to the full joint probability distribution is impractical, as it would require exhaustively modeling all interactions between hardware configurations, compiler optimizations, etc. Instead, by combining Datalog inference with partially known probabilities (e.g., derived from empirical measurements and power consumption models), we enable *quantitative* static analysis that estimates the *severity* of an information leak rather than merely providing a binary vulnerability classification.

3 Overview

Before formalizing our technique in detail, we illustrate how Praline performs probabilistic inference using the synthetic example in Figure 4 that is crafted to give an overview of our approach without being overly complicated. The first six lines declare *input facts*, specifying graph edges, with associated probabilities (e.g., line 1 states that an edge between nodes 5 and 7 exists with probability 0.7). Lines 10-11 are rules that define the notion of paths in the graph. Line 7 states that `edge(1,4)`, `edge(2,5)`, and `edge(2,6)` are statistically correlated, while lines 8-9 specify the known conditional dependencies. For instance, line 8 states that, given `edge(1,4)`, `edge(2,5)`

[2]These probabilities are computed based on the simple Praline program and do not reflect actual probability of heart disease given chest pain.

```
1  0.7::edge(5,7).
2  0.6::edge(1,2).
3  0.8::edge(6,7).
4  0.6::edge(2,5).
5  0.6::edge(1,4).
6  0.6::edge(2,6).
7  corr(edge(1,4),edge(2,5),edge(2,6)).
8  0.80::edge(2,5) | edge(1,4).
9  0.83::edge(2,6) | edge(1,4).
10 path(X,Y) :- edge(X,Y).
11 path(X,Y) :- path(X,Z), edge(Z,Y).
12 query(path(1,7)).
```

Fig. 4. A Praline program

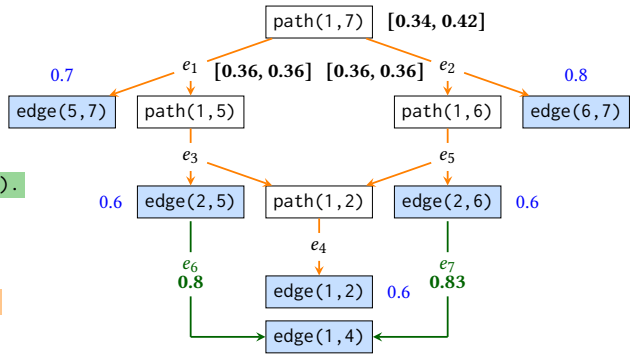


Fig. 5. Derivation graph

exists with probability 0.8. Finally, line 12 queries the probability of `path(1,7)`. We refer to the input facts that are correlated (as declared on line 7) as a *correlation class*.

This example exhibits a salient feature we wish to highlight: Given the information specified by the user, the probability of the predicate `path(1,7)` cannot be determined *exactly*; instead, we can *only* derive upper and lower bounds. In particular, to compute this probability exactly, the user would need to give the full conditional probability table [14] between the predicates `edge(1,4)`, `edge(2,5)`, and `edge(2,6)`; but, in the absence of such information, the only sound conclusion that can be reached is that the probability of `path(1,7)` being true is in the range $[0.34, 0.42]$. To the best of our knowledge, there is no existing technique that can perform precise and accurate probabilistic inference for this type of scenario. In the rest of this section, we illustrate how our proposed inference technique addresses this challenge.

3.1 Basic Approach: Inference via Constrained Optimization

Our starting point is a formulation of this probabilistic inference task as a constrained optimization problem. To formulate this optimization problem, our approach introduces a set of *joint probability variables*, each representing the unknown value of an entry in the conditional probability table for a given correlation class. In our example, the predicates `edge(2,5)`, `edge(1,4)`, and `edge(2,6)` form a correlation class, while `edge(5,7)`, `edge(6,7)`, and `edge(1,2)` are mutually independent. For the singleton correlation class containing `edge(5,7)`, we introduce two joint probability variables, $V_1[1]$ and $V_1[0]$, representing the probabilities of the predicate `edge(5,7)` being true and false, respectively. Since its probability is given as 0.7, the first constraint in Figure 6 sets $V_1[1] = 0.7$ and, by the `SumToOne` constraint, sets $V_1[0] = 0.3$. The constraints for the correlation class with predicates `edge(2,5)`, `edge(1,4)`, and `edge(2,6)` are more involved. Since this class contains 3 predicates, it requires 8 joint probability variables, representing all boolean combinations of these predicates. For example, the variable $V_4[100]$ in Figure 6 represents the probability of `edge(2,5)` being true and `edge(1,4)` and `edge(2,6)` being false. Line 4 of Figure 4 enforces the constraint $V_4[100]+V_4[101]+V_4[110]+V_4[111] = 0.6$, ensuring that the sum of joint probability variables where `edge(2,5)` holds is 0.6. Similarly, the conditional probability from line 8 induces the constraint $V_4[110] + V_4[111] = 0.48$, capturing $P(\texttt{edge(2,5)} \wedge \texttt{edge(1,4)}) = P(\texttt{edge(2,5)}|\texttt{edge(1,4)}) \times P(\texttt{edge(1,4)})$. Figure 6 presents all such constraints for the example in Figure 4.

In addition to generating these constraints, our method also expresses the unknown probability of output facts in terms of the joint probability variables using the logical derivation graph of the input program. In particular, Figure 5 shows the logical derivation of the output predicate `path(1,7)` as

Constraints	
`edge(5,7) = 0.7`	$\mapsto V_1[1] = 0.7$
`edge(1,2) = 0.6`	$\mapsto V_2[1] = 0.6$
`edge(6,7) = 0.8`	$\mapsto V_3[1] = 0.8$
`edge(2,5) = 0.6`	$\mapsto V_4[100] + V_4[101] + V_4[110] + V_4[111] = 0.6$
`edge(1,4) = 0.6`	$\mapsto V_4[010] + V_4[011] + V_4[110] + V_4[111] = 0.6$
`edge(2,6) = 0.6`	$\mapsto V_4[001] + V_4[011] + V_4[101] + V_4[111] = 0.6$
`0.80 :: edge(2,5) | edge(1,4).`	$\mapsto V_4[110] + V_4[111] = 0.8 * 0.6$
`0.83 :: edge(2,6) | edge(1,4).`	$\mapsto V_4[011] + V_4[111] = 0.83 * 0.6$
`SumToOne`	$\mapsto \sum_{b\in\mathbb{B}} V_1[b] = 1 \quad \sum_{b\in\mathbb{B}} V_2[b] = 1 \quad \sum_{b\in\mathbb{B}} V_3[b] = 1 \quad \sum_{b\in\mathbb{B}^3} V_4[b] = 1$
`Input`	$\mapsto \forall b \in \mathbb{B}, V_1[b] \in [0,1],\ V_2[b] \in [0,1],\ V_3[b] \in [0,1], \forall b \in \mathbb{B}^3, V_4[b] \in [0,1]$
Objective	
`path(1,7)`	$\mapsto V_1[1]V_2[1]V_3[1](V_4[111] + V_4[101]) + V_1[1]V_2[1]V_3[0](V_4[110] + V_4[100]) +$ $V_1[1]V_2[1]V_3[1](V_4[110] + V_4[100]) + V_1[1]V_2[1]V_3[0](V_4[111] + V_4[101]) +$ $V_1[0]V_2[1]V_3[1](V_4[011] + V_4[001]) + V_1[0]V_2[1]V_3[1](V_4[101] + V_4[111]) +$ $V_1[1]V_2[1]V_3[1](V_4[011] + V_4[001])$

Fig. 6. Optimization problem for running example

a graph where nodes correspond to predicates and edges correspond to Datalog rule applications. For instance, according to Figure 5, there are two ways to derive the predicate `path(1,7)`: one using `edge(5,7)` and `path(1,5)` and another using `path(1,6)` and `edge(6,7)`. As shown in Figure 6 under `Objective`, our method uses this information to express the probability of predicate `path(1,7)` being true in terms of the joint probability variables. Finally, our method computes the probability bounds for `path(1,7)` being true by minimizing and maximizing the objective function subject to the constraints shown in Figure 6. The bold annotations on the derivation graph in Figure 5 show the probability bounds obtained for each relation using our method. It is worth re-iterating that these probabilities are ranges rather than absolute values *not* because of some imprecision in this solution but rather because of unknown conditional dependencies between some of the input facts. The details of this optimization approach are presented in Section 6.

3.2 Scalable δ-Exact Inference

The approach introduced above guarantees precise lower and upper bounds, but it can be quite expensive due to the complexity of the resulting constrained optimization problem. Thus, to scale our approach to large Datalog programs with complex statistical correlations, we propose a δ-exact algorithm that first computes (loose) approximate probabilities for each output fact and then iteratively tightens these probabilities until they are within δ of the true lower and upper bounds. Our novel δ-exact algorithm can be reduced to three steps, described next.

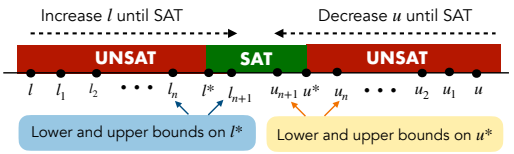


Fig. 7. Iterative strengthening. l^*, u^* are the exact probability bounds; l, u are *approximate* bounds from Step 2, and l_n, u_n are bounds computed by our method.

Step 1: Correlation type inference. Starting with tighter initial approximations accelerates the overall refinement process, and just knowing the *polarity* of the statistical correlation between predicates can help us compute more precise approximations. For example, if two predicates are positively correlated (i.e., one is more likely true if the other is), we can derive tighter bounds than if the correlation type were unknown. Thus, our method first infers *correlation types* between predicates, classifying them as *positive*, *negative*, *independent*, or *unknown*. To do so, we adapt a simpler version of our constrained optimization method to deduce *how* input facts are correlated. For instance, in our example, while the program specifies that `edge(2,5)`, `edge(2,6)`, and `edge(1,4)` are correlated, it does not specify *how*. Using our analysis, we first deduce that `edge(2,5)`, `edge(2,6)` are positively

correlated, which helps establish that `path(1,5)`, `path(1,6)` are *also* positively correlated. We present the details of our correlation analysis method in Section 7.1.

Step 2: Deriving approximate probability bounds. Next, our algorithm uses the correlation analysis results to compute sound but approximate bounds on output probabilities. This involves bottom-up propagation of probabilities from the leaves to the roots of the derivation graph, applying pre-defined approximation rules at each internal node. In our example, to approximate the probability of `path(1,7)`, the algorithm first computes that `path(1,5)` and `path(1,6)` each have a probability of 0.36 as they consist of two independent events, each with a probability of 0.6. It then moves to `path(1,7)`, which is true if either `path(1,5)` $\wedge$ `edge(5,7)` (event e_1) or `path(1,6)` $\wedge$ `edge(6,7)` (event e_2) is true. To compute the exact probability of `path(1,7)`, the conditional probabilities $P(e_1|e_2)$ and $P(e_2|e_1)$ are needed, but are not provided. In the absence of such information, the Fréchet inequality [45] gives the upper bound $P(e_1 \vee e_2) = \min(1, 0.54)$. However, by utilizing the information that `path(1,5)` and `path(1,6)` are positively correlated, we can compute a tighter upper bound of 0.467. Details of this method are in Section 7.2.

Step 3: Iterative refinement. While sound, the probability bounds from the previous step may be far from the true values. Hence, our method iteratively tightens the bounds until they are within a user-specified distance δ of the true bounds. The intuition is to gradually adjust the bounds while checking whether the resultant bound is consistent with the constraint system induced by the program.

As an illustration, consider the output fact `path(1,7)` in our example. Starting from the bounds $[l, u] = [0.288, 0.467]$ computed by the previous step, the iterative refinement step repeatedly attempts to increase l and decrease u by some amount ϵ until the constraint system becomes satisfiable, as shown in Figure 7. When this step terminates, we know that the the true lower (resp. upper) bound is between l_n and l_{n+1} (resp. between u_n and u_{n+1}) in Figure 7. In a second tightening step, the algorithm performs binary search between l_n and l_{n+1} (and between u_n and u_{n+1}) until the two values are within distance δ of each other. Thus, upon termination, it can guarantee that the computed lower and upper bounds are always within δ of the ground truth. Assuming a user-specified $\delta = 0.05$ in our running example, this approach refines the initial approximate bounds for `path(1,7)` from $[0.288, 0.467]$ to $[0.338, 0.417]$, ensuring they lie within δ of the true bounds $[0.34, 0.42]$. The details of our iterative refinement procedure are provided in Section 7.3.

4 Preliminaries

A Datalog program consists of a set of rules R, where each rule is a Horn clause of the form $R(\vec{x}) \text{ :- } \odot R_1(\vec{y}_1), \ldots, \odot R_n(\vec{y}_1)$ where $\odot$ denotes an optional negation operator. We refer to $R(\vec{x})$ as the *head* of the rule and the right hand side as its *body*. Given a rule r, we write $Body^+$ to denote all predicates without a negation symbol in the front and $Body^-$ to denote all predicates that are negated. Finally, we refer to rules without a body as *input facts*. As standard, we can obtain a solution to a Datalog program by first grounding all predicates over the Herbrand universe and then applying the ground rules to a fixed-point. In the rest of this paper, we represent the *solution* to a Datalog program in terms of a *derivation graph*, which shows how each ground output predicate can be derived using ground input predicates.

Definition 1 **(Derivation graph)**. A *derivation graph* for a Datalog program is a hypergraph (V, E) where V is a set of nodes representing ground predicates, and E is a set of directed hyperedges (h, B^+, B^-, r) representing a ground rule r with head h, body $B^+ = Body^+(r)$ and $B^- = Body^-(r)$. The source vertex set of this hyper-edge is $\{h\}$ and the target vertices are $B^+ \cup B^-$. Given a node n, we write $\text{Pred}(n)$ to denote the ground predicate represented by n.

Example 1. In the derivation graph of Figure 5, the edge e_3 has source node `path(1,5)` and target nodes `edge(2,5)` and `path(1,2)`. The quadruple (`path(1,5)`, {`edge(2,5)`, `path(1,2)`}, $\emptyset$, r_2) is used to represent e_3, where r_2 denotes the rule 1 :: `path(1,5)` :– `path(1,2)`, `edge(2,5)`.

Note that different ways of deriving the same ground predicate correspond to multiple hyperedges that share the same source. Given a Datalog program D, we write $\mathcal{G}(D)$ to denote its derivation graph. Intuitively, the derivation graph encodes all possible ways of deriving a ground predicate given the input facts. In the rest of the paper, we assume that the derivation graph is acyclic, as it corresponds to the result of the solver's internal fixed-point computation.[3]

We also define a *flattened* version of the derivation graph that represents each hyperedge with a *set* of regular edges. As we will see later, this flattened representation is useful for understanding basic relationships between output and input predicates.

Definition 2 (**Flattened derivation graph**). Given a derivation graph $G = (V, E)$, Flatten(G) yields a graph $G' = (V', E')$ where $V' = V$ and there exists an edge $(v, v', \sigma) \in E'$ iff there exists a hyperedge $(v, B^+, B^-, r) \in E$ where $v' \in B^+ \cup B^-$ and $\sigma = 1$ if $v' \in B^+$ and $\sigma = -1$ otherwise.

Example 2. Consider hyperedge e_3 from Figure 5. This corresponds to two separate edges e_3^1 and e_3^2 in the flattened derivation graph, where e_3^1 is from `path(1,5)` to `edge(2,5)` and e_3^2 is from `path(1,5)` to `path(1,2)`. Since there are no negated predicates, both edges have a σ value 1.

The flattened derivation graph is useful for determining whether some output facts depend only positively or negatively on an input fact. Given a path $\pi = (v, v_1, \sigma_1), \ldots, (v_n, v', \sigma_n)$ from node v to v' in the flattened derivation graph, we define the polarity of the path as $\Pi_{i=1}^{n} \sigma_i$, denoted Polarity(π). We can then classify dependencies between different output and input facts as follows:

Definition 3 (**Dependence**). Given a Datalog program D with flattened derivation graph G, we say that an output predicate O *depends on* input predicate I, denoted $I \triangleright O$ if there exists a path from O to I in G. We say that O depends positively (resp. negatively) on I iff there exists a path from O to I with a positive (resp. negative) polarity. We write $I \triangleright^+ O$ to denote (logical) positive dependence and $I \triangleright^- O$ for (logical) negative dependence.

The notion of dependence defined above refers to *logical* rather than *statistical* dependence, and it is possible for an output relation to depend both positively *and* negatively on an input fact.

5 PRALINE: Probabilistic Datalog with Correlated Inputs

In this section, we introduce the syntax of Praline, our probabilistic Datalog variant that allows correlated input facts. A pair of input facts I_1, I_2 are correlated iff $P(I_1|I_2) \neq P(I_1)$, and a *correlation class* represents a set of input facts that may be correlated with each other.

Definition 4. **(Praline program)** A Praline program is a tuple (C, R, P_R, P_I) where C is a set of correlation classes, R is a set of rules whose heads are output relations, P_R is a mapping from rules in R to their probabilities, and P_I is a set of (conditional) probabilities about input facts.

Syntax. Borrowing notation from prior work [17, 43], we express rule probabilities in Praline using the notation $p :: R_i$ meaning that the probability of rule $R_i \in R$ is p, and we assume that rules are statistically independent of each other. However, unlike rules, input facts can be statistically dependent. We express such conditional dependencies using the notation $p :: I \mid S$, meaning that the probability of input fact I given input fact *set* S is p. When S is the empty set, such a rule simply declares the probability of an input fact. Finally, we use the notation Class(I) to denote the

[3]The derivation graph may contain auxiliary relations that are used for breaking cycles. We refer the interested reader to prior work for details on this semantics-preserving transformation [20].

Algorithm 1 SOLVE(D)

Input: Datalog program $D = (C, R, P_R, P_I)$
Output: Mapping M from output relations to their probability intervals

```
1: S_o, G ← SolveStandard(D)          ▷ SolveStandard produces output relations S_o and derivation graph G
2: V ← JointProbabilityVars(D)        ▷ Introduce joint probability variables, as discussed in Section 6.1
3: ξ ← GenExprTemplate(V)             ▷ Generate probability expression templates as discussed in Section 6.2
4: φ ← GenConstraint(V, D, ξ)         ▷ Generate constraints φ as discussed in Section 6.3
5: for R_o ∈ S_o do
6:     Φ ← GenObjective(R_o, V, G, ξ) ▷ Generate objective Φ as discussed in Section 6.4
7:     M[R_o] ← Optimize(Φ, φ)        ▷ Use MIP solver to minimize/maximize Φ subject to φ
8: return M
```

correlation class $C_i \in C$ that input fact I belongs to, and we assume that each input fact $I \in C_i$ has an index, denoted $\mathsf{Index}(I, C_i) \in [1, |C_i|]$, that can be used to uniquely identify I within C_i.

Semantics. We define the semantics of PRALINE as an extension of the least-fixed-point semantics of Datalog, incorporating probabilities through a *possible-worlds interpretation*. In particular, each Herbrand model of a PRALINE program represents a deterministic instantiation of probabilistic rules and facts, forming a possible world ω. Given a Praline program D, we define an interpretation of D as a pair (ω, μ) where ω is a possible world and μ is a full joint probability distribution over instantiated rules and input facts in D. We say that (ω, μ) is a model of D, denoted $(\omega, \mu) \models D$ if ω is a possible world of D (under the standard Datalog semantics) and μ is consistent with both ω and the probabilities in D. Then, given such a μ, each possible world ω has an exact probability associated with it, denoted as $P_\mu(\omega)$. Given an output fact O, we can now define $P_\mu(O)$ as follows:

$$P_\mu(O) = \sum_{\omega \in \Omega} P_\mu(\omega) \text{ where } \Omega = \{\omega \mid (\omega, \mu) \models D, \omega \models O\}$$

Finally, given an output fact O, the *set* of possible probabilities of O is given by:

$$P(O) = \{P_\mu(O) \mid \exists \omega.(\omega, \mu) \models D\}$$

We refer interested readers to Appendix A.3 in the Supplementary Material for a more formal treatment of the semantics.

Problem definition. We conclude this section by defining the probabilistic inference problem addressed in the remainder of this paper.

Definition 5. **(Exact probabilistic inference)** Given a PRALINE program $D = (C, R, P_R, P_I)$, the goal of exact probabilistic inference is to produce a mapping from each derived output fact O to a probability interval $[l, u]$ such that $P(O) = \{x \mid l \leq x \leq u\}$.

In Section 6, we provide a method for solving the exact probabilistic inference problem defined above. However, since exact inference is often computationally intractable, we also introduce the notion of δ-exact inference, which we address in Section 7:

Definition 6. **(δ-exact probabilistic inference)** Given a PRALINE program $D = (C, R, P_R, P_I)$, the goal of δ-exact probabilistic inference is to produce a mapping from each derived output fact O to a probability interval $[l, u]$ such that $l \geq l^* - \delta$ and $u \leq u^* + \delta$ where $P(O) = \{x \mid l^* \leq x \leq u^*\}$.

6 Exact Probabilistic Inference via Constrained Optimization

In this section, we address the *exact* probabilistic inference problem using constrained optimization. As summarized in Algorithm 1, our method consists of five steps. First, for each correlation class C_i, it introduces a set of variables that express the joint probabilities of input facts in that class C_i

(line 2). Second, at line 3, it generates a *probability expression template* ξ over these joint probability variables — the key idea is to express the probabilities of all relations symbolically as instantiations of the same shared template. Then, at line 4, it invokes GenConstraint to express requirements on the joint probability variables based on the facts in the Datalog program. Finally, for each fact R_o, the algorithm first invokes GenObjective (line 6) to express the probability of R_o symbolically as an instantiation Φ of ξ and then optimizes Φ (line 7) subject to the constraints generated earlier.

6.1 Joint Probability Variables

The variables in our encoding represent *joint probabilities* over input facts; hence, we refer to them as *joint probability variables*. For each correlation class C, we introduce a map V_C of joint probability variables, denoted $\mathsf{Rep}(C)$ (for "representative"). Given a correlation class C of size n, V_C contains 2^n variables, one for each possible boolean assignment to input facts in C. We use bitvectors to represent boolean assignments and write $V_C[b]$ to denote the corresponding joint probability variable. For example, for $C = \{I_1, I_2, I_3\}$, $V_C[001]$ denotes the joint probability of I_1, I_2 being false and I_3 being true. We also write $\mathbb{B}^n$ to denote the set of all bitvectors of size n. Given an input predicate I and a joint probability variable v, $v \models I$ (resp. $v \not\models I$) indicates that v represents an event in which I is true (resp. false). In our example, we have $V_C[001] \models I_3$ and $V_C[001] \not\models I_1$.

6.2 Probability Expressions and Templates

As explained earlier, a key idea underlying our algorithm is to express the probability of each output relation as a symbolic expression over the joint probability variables. Because all of these expressions are instantiations of the same template, we first explain what these templates look like.

Definition 7. **(Product term)** Let $C_1 \ldots C_n$ be the set of all correlation classes, i.e., $C_i \in \boldsymbol{C}$. A product term ψ is a product of joint probability variables $v_1 \ldots v_n$ where each v_i is a joint probability variable associated with class C_i, i.e., $v_i = \mathsf{Rep}(C_i)[b]$ for some bitvector b.

Intuitively, a product term represents the probability of a particular truth assignment to *all* input relations in the program. For example, if we have two correlation classes $C_1 = \{I_1, I_2\}$ and $C_2 = \{I_3, I_4\}$, then the product term $\psi = V_1[10] \times V_2[11]$ represents the probability of I_2 being false and I_1, I_3, I_4 being true. Extending our previous notation, given a product term ψ and input relation I, we write $\psi \models I$ (resp. $\psi \not\models I$) to denote that ψ represents the probability of an event in which I is true (resp. false). For instance, in our previous example, we have $\psi \models I_1$ and $\psi \not\models I_2$.

Definition 8. **(Probability expression template)** Let Ψ be the set of all possible product terms. A *probability expression template* ξ is a sum-of-product expression of the form $\sum_{\psi_i \in \Psi} \square_i \times \psi_i$.

For instance, if we have two correlation classes each with a single input fact, the probability expression template would be of the form:

$$\square_1 \times V_1[0]V_2[0] + \square_2 \times V_1[0]V_2[1] + \square_3 \times V_1[1]V_2[0] + \square_4 \times V_1[1]V_2[1]$$

Intuitively, the probability of every relation in the program can be expressed symbolically as an instantiation of a probability expression template, where holes $\square$ are filled by *coefficient terms* λ:

Definition 9. **(Coefficient term)** A coefficient term λ is an expression of the form $\sum_i \lambda_i^+ \times \lambda_i^-$ where $\lambda_i^+ = r_1 \times \ldots \times r_n$ and $\lambda_i^- = (1 - r_1') \times \ldots \times (1 - r_k')$ and each r_j, r_j' is a *variable* representing the probability of some rules in the Datalog program.

A coefficient term represents the probability of an output fact being derived by applying a specific sequence of rules. However, since rule bodies can include negations, the derivation of some facts may depend on certain rules *not* being applied. As a result, coefficient terms also incorporate factors of the form $(1 - r)$, which represent the probability of a rule *not* being applied.

$$
\begin{array}{lcll}
[\![D]\!] & = & \bigwedge_{C_i \in C} [\![C_i]\!] \ \wedge \ \bigwedge_{I_r \in P_I} [\![I_r]\!] \ \wedge \ \bigwedge_{r \in R} \mathsf{ProbVar}(r) = P_R(r) & \\
[\![C]\!] & = & \forall b \in \mathbb{B}^{|C|}.\ \mathsf{Rep}(C)[b] \in [0,1] \ \wedge \ \sum_{b \in \mathbb{B}^{|C|}} \mathsf{Rep}(C)[b] = 1 & \\
[\![I_r]\!] & = & \mathsf{Expr}(I) = p & \text{if} \quad I_r = (p :: I | \varnothing) \\
[\![I_r]\!] & = & \otimes_{i=0}^{n} \mathsf{Expr}(I_i) = p \times \otimes_{i=1}^{n} \mathsf{Expr}(I_i) & \text{if} \quad I_r = (p :: I_0 \mid I_1, \ldots, I_n)
\end{array}
$$

Fig. 8. Rules for producing constraints from a Datalog program $D = (C, R, P_R, P_I)$. ProbVar(r) is a fresh variable representing the probability of rule r.

Definition 10. **(Template instantiation)** An *instantiation* of template ξ either replaces each hole with a $c \in \{0, 1\}$ or with a coefficient term λ. Given a vector of expressions σ, we use the notation $\xi[\sigma]$ to denote a probability expression that is obtained by filling hole $\square_i$ in ξ with expression σ_i.

Intuitively, template instantiations that fill holes with coefficient terms represent probabilities of output facts whereas those that use constants represent probabilities of input relations. Given a relation R, $\mathsf{Expr}(R)$ denotes the symbolic expression representing the probability of R and is obtained through some instantiation σ of the holes in ξ — i.e., $\mathsf{Expr}(R) = \xi[\sigma]$ for some σ.

Definition 11. **(Input expression)** Let ξ be a template with ψ_i as its i'th term. The *probability expression* for input fact I, denoted $\mathsf{Expr}(I)$, is given by $\sigma_i = 1$ if $\psi_i \models I$ and $\sigma_i = 0$ otherwise.

In other words, the symbolic expression for an input relation is obtained by choosing 1 as the coefficient of a product term ψ_i if ψ_i represents the probability of an event in which I is true and 0 otherwise. It is easy to see that $\mathsf{Expr}(I)$ symbolically represents the probability of an input relation in terms of the joint probability variables.

6.3 Constraint Generation

We now turn to the GenConstraint procedure used in Algorithm 1, with its implementation summarized in Figure 8. Given a program $D = (C, R, P_R, P_I)$, this procedure generates three types of constraints. First, for each correlation class C_i with representative V_i, it introduces a constraint ensuring that all variables in V_i are valid probabilities in the range $[0, 1]$ and that they sum to 1 (line 2 of Figure 8). Second, the procedure introduces constraints to encode the (conditional) input probabilities in the Datalog program. Specifically, for each rule of the form $p :: I$, a constraint is introduced stating that $\mathsf{Expr}(I) = p$, where $\mathsf{Expr}(I)$ is defined in Definition 11. Additionally, for conditional probability declarations of the form $p :: I_0 | I_1 \ldots I_n$ (line 4 of Figure 8), a constraint is introduced stating that $P(I_0 \wedge \ldots \wedge I_n) = p \times P(I_1 \wedge \ldots \wedge I_n)$. Note that the probability $P(I_0 \wedge \ldots \wedge I_n)$ is computed using a special multiplication operator $\otimes$, which is explained in the next section. Finally, the procedure generates a third type of constraint that relates variables in the coefficient terms to the actual rule probabilities given by P_R (the last conjunct in the first line).

6.4 Generating Optimization Objective

In this final subsection, we focus on the GenObjective procedure. As a reminder, the purpose of this procedure is to express the probability of each output relation as a symbolic expression over the joint probability variables—i.e., as an instantiation of the probability expression template defined in Definition 10. The key idea behind this procedure is to use the logical derivation graph G to find a coefficient term that can be used to fill each hole in the expression template. In particular, the algorithm performs bottom-up traversal of the derivation graph to construct a symbolic probability expression of each node, utilizing the probability expressions of its children.

Our bottom-up traversal algorithm is presented in Figure 9. The base case is the rule labeled Leaf, which corresponds to input relations. Since we already know how to compute the probability expression for an input relation (see Definition 11), the Leaf rule simply produces $\mathsf{Expr}(I)$, where

$$\frac{v \in \text{Leaves}(G) \quad \text{Pred}(v) = I}{G \vdash v \hookrightarrow \text{Expr}(I)} \text{ (Leaf)}$$

$$\frac{\text{OutgoingEdges}(G, v) = \{e_1, \ldots, e_n\} \qquad G \vdash e_i \hookrightarrow E_i \text{ for } i \in [1, n] \qquad E_1 \oplus \cdots \oplus E_n \rightsquigarrow E}{G \vdash v \hookrightarrow E} \text{ (Node)}$$

$$\frac{\begin{array}{c} B^+ = \{v_1^+, \ldots v_m^+\} \quad G \vdash v_i^+ \hookrightarrow E_i^+ \text{ for } i \in [1, m] \qquad B^- = \{v_1^-, \ldots v_n^-\} \qquad G \vdash v_i^- \hookrightarrow E_i^- \text{ for } i \in [1, n] \\ (E_1^+ \otimes \cdots \otimes E_m^+) \rightsquigarrow E^+ \qquad (\ominus(E_1^-)) \otimes \cdots \otimes (\ominus(E_n^-)) \rightsquigarrow E^- \quad E^+ \otimes E^- \rightsquigarrow E \end{array}}{G \vdash (\_, B^+, B^-, r) \hookrightarrow \text{ProbVar}(r) \times E} \text{ (Edge)}$$

Fig. 9. Rules for inferring symbolic probability expression for output relations.

$$\frac{E = \sum \lambda_i \psi_i}{\ominus(E) \rightsquigarrow \sum (1 - \lambda_i)\psi_i} \text{ (Neg)} \qquad \frac{E_1 = \sum \lambda_{1i}\psi_i \quad E_2 = \sum \lambda_{2i}\psi_i \quad \text{JointProb}(\lambda_{1i}, \lambda_{2i}) \rightsquigarrow \lambda_i}{\vdash E_1 \otimes E_2 \rightsquigarrow \sum \lambda_i \psi_i} \text{ (Mul)}$$

$$\frac{E_1 = \sum \lambda_{1i}\psi_i \quad E_2 = \sum \lambda_{2i}\psi_i \quad \text{JointProb}(\lambda_{1i}, \lambda_{2i}) \rightsquigarrow \lambda_i}{E_1 \oplus E_2 \rightsquigarrow \sum (\lambda_{1i} + \lambda_{2i} - \lambda_i)\psi_i} \text{ (Add)}$$

$$\frac{\lambda_1 = \sum_{i=1}^{n} e_i \quad \lambda_2 = \sum_{j=1}^{m} e_j \quad \forall (i,j).\, e_i \diamond e_j \rightsquigarrow e_{ij}}{\text{JointProb}(\lambda_1, \lambda_2) \rightsquigarrow \sum_{(i,j)=1}^{(n,m)} e_{ij}} \text{ (Joint)} \qquad \frac{\begin{array}{c} e_i = X_i^+ \times X_i^- \quad e_j = Y_j^+ \times Y_j^- \\ X_i^+ \blacklozenge Y_j^+ \rightsquigarrow Z^+ \quad X_i^- \blacklozenge Y_j^- \rightsquigarrow Z^- \end{array}}{e_i \diamond e_j \rightsquigarrow \text{Disjoint}(X_i, Y_j)\ ?\ Z^+ Z^-\ :\ 0} \text{ ($\diamond$)}$$

$$\frac{}{\prod_{i=1}^{p} \bigcirc(x_i) \blacklozenge \prod_{j=1}^{q} \bigcirc(y_j) \rightsquigarrow \prod_{z_k \in (\bigcup_{i=1}^{p} x_i \cup \bigcup_{j=1}^{q} y_j)} \bigcirc(z_k)} \text{ ($\blacklozenge$)}$$

Fig. 10. Rules defining $\ominus, \otimes$ and $\oplus$ operations. $\text{Disjoint}(X_i, Y_j)$ is true iff $\text{Vars}(X_i^+) \cap \text{Vars}(Y_j^-) = \emptyset \wedge \text{Vars}(X_i^-) \cap \text{Vars}(Y_j^+) = \emptyset$. $\text{Vars}(X)$ denotes the set of rule probability variables present in X. In the $\blacklozenge$ rule, $\bigcirc(x)$ represents a term of the form x or $1 - x$.

I is the input relation represented by the node v. Next, the rule labeled Node describes how to compute the probability expression for an internal node v representing an output relation. If a node labeled O has n outgoing edges, this means O can be derived in n different ways. Therefore, the rule first computes a probability expression E_i for each edge e_i, representing the probability of a possible derivation of O. It then computes the probability of $P(e_1 \vee \ldots \vee e_n)$ using a special $\oplus$ operator, which we explain later. Finally, the last rule, labeled Edge, computes the probability expression for a derivation. Recall that a hyperedge with source v and target nodes B^+ and B^- represents the application of a rule r, whose body consists of positive facts B^+ and negative facts B^-. This rule first computes the probability expressions E_i^+ for each $v_i^+ \in B^+$ and E_i^- for each $v_i^- \in B^-$, respectively. It then combines these to obtain a probability expression for the edge.

To combine these expressions, we first note that the probability expression for $\neg R$ is given by $\ominus E$, where E denotes the probability expression for R, i.e., $E = \text{Expr}(R)$. Next, we define an operator $\otimes$ (explained later) to compute the probability of $P(v_1 \wedge v_2)$ as $P(v_1) \otimes P(v_2)$. Therefore, the Edge rule first computes the probability of all positive facts being true as $E^+ = E_1^+ \otimes \ldots \otimes E_m^+$, and the probability of all negative facts being true as $E^- = \ominus(E_1^-) \otimes \ldots \otimes \ominus(E_n^-)$. The probability of both positive and negative facts being satisfied is then given by $E = E^+ \otimes E^-$. Finally, since the rule itself has a probability, the final probability expression is obtained as $\text{ProbVar}(r) \times E$, where $\text{ProbVar}(r)$ is the variable representing the probability of rule r.

Finally, we turn our attention to the definitions of the $\ominus$, $\otimes$, and $\oplus$ operators from Figure 10, which are used to compute probability expressions for conjunctions and disjunctions of events.

$p_1 :: I_1$	$Pr(I_1)$	$= 0 * V[00] + 0 * V[01] + 1 * V[10] + 1 * V[11]$
$p_2 :: I_2$	$Pr(I_2)$	$= 0 * V[00] + 1 * V[01] + 0 * V[10] + 1 * V[11]$
$p_3 :: I_2 \vert I_1$		
$r_1 :: A :- I_1$	$Pr(A)$	$= 0 * V[00] + 0 * V[01] + r_1 * V[10] + r_1 * V[11]$
$r_2 :: B :- A$	$Pr(B)$	$= 0 * V[00] + 0 * V[01] + r_1 r_2 * V[10] + r_1 r_2 V[11]$
$r_3 :: C :- \backslash +A, I_2$	$Pr(\backslash +A)$	$= 1 * V[00] + 1 * V[01] + (1 - r_1) V[10] + (1 - r_1) V[11]$
	$Pr(C)$	$= r_3 \times (Pr(\backslash +A) \otimes Pr(I_2)) = 0 * V[00] + r_3 * V[01] + 0 * V[10] + r_3 (1 - r_1) V[11]$
$r_4 :: D :- B, A$	$Pr(D)$	$= r_4 \times (Pr(B) \otimes Pr(A)) = 0 * V[00] + 0 * V[01] + r_1 r_2 r_4 V[10] + r_1 r_2 r_4 V[11]$
$r_5 :: E :- C$	$Pr(E_1)$	$= 0 * V[00] + r_3 r_5 * V[01] + 0 * V[10] + r_3 r_5 (1 - r_1) V[11]$
$r_6 :: E :- D$	$Pr(E_2)$	$= 0 * V[00] + 0 * V[01] + r_1 r_2 r_4 r_6 V[10] + r_1 r_2 r_4 r_6 V[11]$
	$Pr(E)$	$= Pr(E_1) \oplus Pr(E_2)$
		$= 0 * V[00] + r_3 r_5 V[01] + r_1 r_2 r_4 r_6 V[10] + (r_3 r_5 (1 - r_1) + r_1 r_2 r_4 r_6 - 0) V[11]$

Fig. 11. Left: Praline program. Right: Probability expressions for both input and output facts.

All of these rules rely on the fact that probability expressions are in a normalized form, consisting of sums of terms of the form $\lambda \times \psi$, where λ is a coefficient term (Def. 9) and ψ is a product term (Def. 7). First, given a symbolic expression E representing the probability of some relation R, the Neg rule computes the probability of $\neg R$ by simply replacing all coefficient terms λ with $1 - \lambda$. Second, the Mul rule computes the probability of a conjunction of events by combining the coefficients of each product term using the JointProb function (explained later). Similarly, the Add rule computes the probability of a disjunction of events by updating the coefficient of each term ψ_i as $\lambda_{1i} + \lambda_{2i} - \text{JointProb}(\lambda_{1i}, \lambda_{2i})$. Intuitively, this corresponds to an application of the inclusion-exclusion principle.

Next, we focus on the last three rules for computing the joint probability of two coefficient terms λ_1 and λ_2. The rule labeled Joint essentially distributes multiplication over addition, as we require each coefficient term to be a sum of products. The second rule, labeled $\diamond$, considers multiplication expressions of the form $(X_i^+ \times X_i^-)$ and $(Y_i^+ \times Y_i^-)$. In this rule, Z^+ denotes a product of variables v, and Z^- represents a product of terms of the form $1 - v$, where v represents a rule probability. Intuitively, Z^+ indicates the rules that must be applied to derive a fact, while Z^- indicates the rules that must not be applied. Thus, if a variable v appears in X_i^+ (or Y_i^+) and $1 - v$ appears in Y_j^- (or X_i^-), this results in a contradiction, and the probability of the term is zero. Otherwise, if the disjointness condition is satisfied, the probability is computed using the rule labeled $\blacklozenge$. The intuition behind the $\blacklozenge$ rule is as follows: If a variable v appears in both X_i and Y_i, it indicates that the same fact is derived using the same rule. To avoid overcounting, we do not multiply probabilities repeatedly, as that rule only needs to be applied once. Therefore, the $\blacklozenge$ rule multiplies probabilities after ensuring that variables associated with the same rule are treated uniquely.

Example 3. Consider the Praline program on the left side of Figure 11, with the corresponding probability expressions for both input and output facts displayed on the right. In this example, I_1 and I_2 are correlated, so they belong to the same correlation class V — e.g., $V[01]$ represents the joint probability of I_1 being false and I_2 being true. The right side of Figure 11 shows the symbolic probability expressions for each fact. The parts highlighted in red show the need for the disjointness check in the $\diamond$ rule; and the parts highlighted in blue illustrate the need for the $\blacklozenge$ operator.

Theorem 1. *Let E_1 and E_2 denote the probability expressions of events A and B respectively. Then, we have: (1) If $\ominus E_1 \rightsquigarrow E$, then E represents the probability of $\neg A$; (1) if $E_1 \oplus E_2 \rightsquigarrow E$, then E represents the probability of event $A \vee B$ (3) iff $E_1 \otimes E_2 \rightsquigarrow E$,then E represents the probability of event $A \wedge B$.*

7 δ-Exact Probabilistic Inference

Building on our approach from Section 6, we now present a more practical δ-exact probabilistic inference algorithm, summarized in Algorithm 2. Similar to the previous algorithm, it first uses the

Algorithm 2 $\textsc{Solve}^{\pm\delta}(D, \delta)$

Input: Datalog program D, precision bound δ
Output: Mapping M that maps output facts to probabilities

1: $S_o, G \leftarrow$ `SolveStandard`(D) ▷ `SolveStandard` produces output facts S_o and derivation graph G
2: $\phi \leftarrow \textsc{GenConstraints}(D, G)$ ▷ Use technique from Section 6.3 to generate constraints
3: $\Sigma \leftarrow \textsc{InferCorrelationType}(G, \phi)$ ▷ Infer type of statistical correlation between facts
4: $M \leftarrow \textsc{DeriveApproximateBounds}(G, D, \Sigma)$ ▷ Approximate bounds using static analysis and constraint solving
5: $M' \leftarrow \textsc{MakeDeltaPrecise}(M, S_o, \phi, \delta)$ ▷ Iterative refinement of bounds
6: **return** M'

$$\frac{p :: (I|\varnothing) \in P_I \quad p < 1}{D, \phi \vdash I \blacktriangleright^{+} I}\ (\textsc{Id}) \qquad \frac{\mathsf{Class}(I_1) \neq \mathsf{Class}(I_2)}{D, \phi \vdash I_1 \blacktriangleright^{\perp} I_2}\ (\textsc{Indep}) \qquad \frac{I_1 \blacktriangleright^{\star} I_2 \ \ (\star \in \{+, -, \perp\})}{D, \phi \vdash I_2 \blacktriangleright^{\star} I_1}\ (\textsc{Symm})$$

$$\frac{E_1 = \mathsf{Expr}(I_1) \quad E_2 = \mathsf{Expr}(I_2) \quad E_1 \otimes E_2 \rightsquigarrow E_\wedge \quad \models \phi \Rightarrow E_\wedge \boxdot E_1 \times E_2 \quad \star = \mathsf{Sign}(\boxdot)}{D, \phi \vdash I_1 \blacktriangleright^{\star} I_2}\ (\textsc{Semantic})$$

Fig. 12. Inference of statistical correlation between input variables. Here, $\boxdot \in \{<, >, =\}$, and $\mathsf{Sign}(\boxdot)$ yields + for $>$, − for $<$, and $\perp$ for $=$.

underlying Datalog solver to obtain a derivation graph G. Next, it invokes the InferCorrelationType function to compute the *type* of the statistical correlation between facts, where a correlation type is either positive, negative, independent, or unknown. In the third step, the algorithm uses this correlation type environment Σ to derive *approximate* probability bounds on output facts. Finally, the call to MakeDeltaPrecise at line 5 keeps refining the inferred bounds until the derived bounds are within δ of the ground truth.

7.1 Inference of Correlation Types

In this section, we present the InferCorrelationType algorithm that can be used to infer whether a pair of relations are positively/negatively correlated or whether they are independent.

Definition 12. **(Statistical correlation)** Two events X and Y are positively (resp. negatively) correlated if $P(X|Y) > P(X)$ (resp. $P(X|Y) < P(X)$). If $P(X|Y) = P(X)$, then X and Y are independent.

Note that the notion of statistical correlation is symmetric. That is, if X is positively correlated with Y, then Y is also positively correlated with X. A *correlation type* for a pair of Datalog facts is one of Pos (+), Neg (−), $\perp$, $\top$, where $\perp$, $\top$ indicate independence and unknown correlation respectively. As stated earlier, identifying correlation types allows us to derive tighter approximate bounds than would otherwise be possible. To infer these correlation types, our method proceeds in two phases: First, it infers statistical correlations between input facts. In the second phase, it uses this information to infer statistical correlations between outputs.

Phase 1: Inferring correlation types between input facts. Figure 12 presents our method for inferring correlation types for pairs of input facts using the judgment $D, \phi \vdash I_1 \blacktriangleright^{\star} I_2$ where $\star \in \{+, -, \perp\}$ indicates positive and negation correlation and statistical independence respectively. Here, D is the Datalog program and ϕ is the set of constraints generated from the Datalog program as described in Section 6.3. The first rule, labeled Id indicates that the input fact I is positively correlated with itself. The next rule, labeled Indep applies to two input facts that do not belong to the same correlation class. Finally, the last rule labeled Semantic uses the constraint-based technique from Section 6 to check for statistical correlation. The basic idea is to use the algorithm

$$
\begin{array}{lcllcl}
\mathrm{Dep}(O) & = & \{I \mid I \triangleright O\} & \mathrm{Dep}^{\star}(O) & = & \{I \mid I \triangleright^{\star} O\} \\
\mathrm{Dep}(E_1 \wedge E_2) & = & \mathrm{Dep}(E_1) \cup \mathrm{Dep}(E_2) & \mathrm{Dep}^{\star}(E_1 \wedge E_2) & = & \mathrm{Dep}^{\star}(E_1) \cup \mathrm{Dep}^{\star}(E_2) \\
\mathrm{Dep}(E_1 \vee E_2) & = & \mathrm{Dep}(E_1) \cup \mathrm{Dep}(E_2) & \mathrm{Dep}^{\star}(E_1 \vee E_2) & = & \mathrm{Dep}^{\star}(E_1) \cup \mathrm{Dep}^{\star}(E_2) \\
\mathrm{Dep}(\neg E) & = & \mathrm{Dep}(E) & \mathrm{Dep}^{\star}(\neg E) & = & \mathrm{Dep}^{\overline{\star}}(E)
\end{array}
$$

Fig. 13. Auxiliary Dep function. O denotes an output fact and $\star \in \{+, -\}$. $\overline{\star}$ is + if $\star$ is − and vice versa.

$$
\frac{\forall (x, y) \in \mathrm{Dep}(E).\ x \neq y \rightarrow \mathrm{Class}(x) \neq \mathrm{Class}(y)}{\vdash \chi(E)}\ (\textsc{NonInterfere})
$$

$$
\frac{\exists (x, y) \in \mathrm{Dep}^{\star}(E_1) \times \mathrm{Dep}^{\star}(E_2).\ x \blacktriangleright^{\natural} y}{\vdash E_1 \rightharpoonup^{\natural} E_2}\ (\textsc{May-1}) \qquad \frac{\exists (x, y) \in \mathrm{Dep}^{\star}(E_1) \times \mathrm{Dep}^{\overline{\star}}(E_2).\ x \blacktriangleright^{\natural} y}{\vdash E_1 \rightharpoonup^{\overline{\natural}} E_2}\ (\textsc{May-2})
$$

$$
\frac{\vdash \chi(E_1) \quad \vdash \chi(E_2) \quad \vdash E_1 \rightharpoonup^{+} E_2 \quad \nvdash E_1 \rightharpoonup^{-} E_2}{\vdash E_1 \xrightarrow{\oplus} E_2}\ (\textsc{Pos}) \qquad \frac{\vdash \chi(E_1) \quad \vdash \chi(E_2) \quad \vdash E_1 \rightharpoonup^{-} E_2 \quad \nvdash E_1 \rightharpoonup^{+} E_2}{\vdash E_1 \xrightarrow{\ominus} E_2}\ (\textsc{Neg})
$$

$$
\frac{\forall x, y \in \mathrm{Dep}(E_1) \times \mathrm{Dep}(E_2).\ \mathrm{Class}(x) \neq \mathrm{Class}(y)}{\vdash E_1 \xrightarrow{\perp} E_2}\ (\textsc{Indep}) \qquad \frac{\nvdash E_1 \xrightarrow{\oplus} E_2 \quad \nvdash E_1 \xrightarrow{\ominus} E_2 \quad \nvdash E_1 \xrightarrow{\perp} E_2}{\vdash E_1 \xrightarrow{\top} E_2}\ (\textsc{Unknown})
$$

Fig. 14. Inference rules for computing statistical correlation between expressions, where $\natural \in \{+, -\}$. Note that predicates $x \blacktriangleright^{\star} y$ are derived using Figure 12, and $\overline{\star}$ (resp. $\overline{\natural}$) is + if $\star$ (resp. $\natural$) is − and vice versa.

of Section 6 to check how the joint probability of $I_1 \wedge I_2$ relates to the product of the individual probabilities of I_1 and I_2. To do so, it first generates symbolic expressions $E_1, E_2, E_\wedge$ for I_1, I_2, and $I_1 \wedge I_2$ respectively. Then, it uses a solver to check whether the constraints ϕ (encoding the input probabilities) logically imply whether $E_\wedge \boxdot E_1 \times E_2$, where $\boxdot$ denotes one of $<, >, =$. Note that this semantic approach is not as susceptible to the scalability challenges discussed earlier because we consider *only* input facts and *only* those that belong to the same correlation class.

Theorem 2. *Suppose that we derive $I_1 \blacktriangleright^{\star} I_2$ using the rules from Figure 12. Then, $p(I_1|I_2) > p(I_1)$ if $\star = +$, $p(I_1|I_2) < p(I_1)$ if $\star = -$, and $p(I_1|I_2) = p(I_1)$ if $\star = \perp$.*

Phase 2: Inferring correlation types between outputs. The second phase of our inference algorithm uses the results of the first phase to infer correlation types between expressions involving outputs. Note that we could, *in principle*, use the same constraint-based approach presented Figure 12 to infer correlation types between arbitrary expressions; however, such an approach does not scale well. To overcome this scalability bottleneck, we instead utilize lightweight static analysis.

The key idea underlying our method is to utilize the derivation graph, along with the known correlations between input facts, to infer correlation types between arbitrary expressions (i.e., boolean combinations of ground predicates). Given a pair of expressions E_1, E_2, our method first uses the derivation graph to identify the set S_1, S_2 of input facts, along with their polarity, that E_1 and E_2 *logically* depend on; it then analyzes the *statistical* correlations between elements in S_1, S_2 to decide whether we can determine the correlation type between E_1 and E_2.

Our analysis is summarized in Figures 13 and 14, where the former defines two auxiliary functions $\mathrm{Dep}, \mathrm{Dep}^{\star}(\star \in \{+, -\})$ used in Figure 14. As shown in Figure 13, $\mathrm{Dep}(E)$ simply yields the set of all input facts that E is *logically* dependent on according to the derivation graph (recall Def 3). Similarly, $\mathrm{Dep}^{+}(E)$ (resp. $\mathrm{Dep}^{-}(E)$) yields the set of input facts that E depends *positively* (resp. *negatively*) on. For example, consider an output fact O that can be derived using $I_1 \wedge \neg I_2$ or using only I_2. In this case, both $\mathrm{Dep}(O)$ and $\mathrm{Dep}^{+}(O)$ include I_1, I_2 but $\mathrm{Dep}^{-}(O)$ only includes I_2.

Figure 14 uses these auxiliary functions to infer correlation types between arbitrary expressions. The basic idea is to infer whether two expressions E_1 and E_2 *may* be positively or negatively

Algorithm 3 DeriveApproximateBounds(G, D, Σ)

1: **for** node $n \in$ InternalNodes(G) **do**
2: $\quad E_n \leftarrow [\![n]\!](G)$
3: $\quad M[n] \leftarrow$ ApproxExpr(Σ, D, E_n)
4: **return** M

$$
\begin{array}{lcll}
[\![n]\!](G) & = & \text{Pred}(n) & \text{if } n \text{ is a leaf node} \\
[\![n]\!](G) & = & \bigvee_{e \in E}([\![e]\!](G), \mathbb{P}(e)) & \text{where OutEdges}(G, n) = E \\
[\![e]\!](G) & = & \bigwedge_{n \in B^+}[\![n]\!](G) & \text{where } e = (n_s, B^+, B^-, r) \\
 & & \wedge \bigwedge_{n \in B^-} \neg [\![n]\!](G) &
\end{array}
$$

Fig. 15. Computation of derivation expressions. $\mathbb{P}(e)$ yields the probability of the rule labeling edge e.

Table 1. CL/CU/DL/DU computation rules. CL and CU denote the lower and upper bounds of the conjunction operation, respectively, while DL and DU represent the lower and upper bounds of the disjunction operation.

Operation	$\star = +$	$\star = -$	$\star = \bot$	$\star = \top$
$\text{CL}(e_1, e_2, \star)$	$e_1 \times e_2$	$\max(e_1 + e_2 - 1, 0)$	$e_1 \times e_2$	$\max(e_1 + e_2 - 1, 0)$
$\text{CU}(e_1, e_2, \star)$	$\min(e_1, e_2)$	$e_1 \times e_2$	$e_1 \times e_2$	$\min(e_1, e_2)$
$\text{DL}(e_1, e_2, \star)$	$\max(e_1, e_2)$	$1 - (1 - e_1)(1 - e_2)$	$1 - (1 - e_1)(1 - e_2)$	$\max(e_1, e_2)$
$\text{DU}(e_1, e_2, \star)$	$1 - (1 - e_1)(1 - e_2)$	$\min(1, e_1 + e_2)$	$1 - (1 - e_1)(1 - e_2)$	$\min(1, e_1 + e_2)$

$$
\frac{I \in \text{InputFacts}(D) \quad p :: (I \mid \varnothing) \in \text{InputProbs}(D)}{\Sigma, D \vdash I \rightsquigarrow [p, p]}\ (\textsc{In}) \qquad \frac{\Sigma, D \vdash E \rightsquigarrow [l, u]}{\Sigma, D \vdash \neg E \rightsquigarrow [1 - u, 1 - l]}\ (\textsc{Neg})
$$

$$
\frac{\Sigma, D \vdash E_1 \rightsquigarrow [l_1, u_1] \quad \Sigma, D \vdash E_2 \rightsquigarrow [l_2, u_2] \quad \Sigma(E_1, E_2) = \star}{\Sigma, D \vdash E_1 \wedge E_2 \rightsquigarrow [\text{CL}(l_1, l_2, \star), \text{CU}(u_1, u_2, \star)]}\ (\textsc{Conjunct})
$$

$$
\frac{\Sigma, D \vdash E_1 \rightsquigarrow [l_1, u_1] \quad \Sigma, D \vdash E_2 \rightsquigarrow [l_2, u_2] \quad \Sigma(E_1, E_2) = \star}{\Sigma, D \vdash (E_1, p_1) \vee (E_2, p_2) \rightsquigarrow [\text{DL}(l_1 \times p_1, l_2 \times p_2, \star), \text{DU}(u_1 \times p_1, u_2 \times p_2, \star)]}\ (\textsc{Disjunct})
$$

Fig. 16. Inference rules for computing approximate probability bounds.

correlated (rules labeled May), meaning that a pair of shared input predicates in E_1 and E_2 have the potential to introduce a positive or negative correlation. Then, according to the rules labeled Pos and Neg, if we find that E_1 and E_2 may be positively (resp. negatively) correlated and there is nothing that introduces a potential negative (resp. positive) correlation, we can conclude that E_1 and E_2 are definitely positively (resp. negatively) correlated as long as both expressions exhibit a certain non-interference property shown in the NonInterfere rule as $\chi(E)$. Intuitively, the non-interference property is necessary because, if an input fact I is positively correlated with I_1 and I_2 individually, it does not necessarily mean that it is positively correlated with $I_1 \wedge I_2$. At the end of the correlation type analysis, the inferred dependencies are stored in Σ and used in Algorithm 2.

Theorem 3. *If $E_1 \xrightarrow{\oplus} E_2$ is derivable using the rules in Figure 14, then $p(E_1|E_2) > p(E_1)$. Similarly, $\vdash E_1 \xrightarrow{\ominus} E_2$ implies $p(E_1|E_2) < p(E_1)$ and $\vdash E_1 \xrightarrow{\perp} E_2$ implies $p(E_1|E_2) = p(E_1)$.*

7.2 Computing Approximate Probability Bounds

In this section, we present a technique, summarized in Algorithm 3, for deriving *approximate* probability bounds on output relations. For each node in the derivation graph, this algorithm computes a so-called *derivation expression* E that summarizes all ways in which a given relation can be derived (line 2). For example, if there are two rules $p_1 :: R :- A, B$ and $p_2 :: R :- C$, then the derivation expression is of the form $(A \wedge B) \vee C$. However, because we also need to keep track of the rule probabilities, expressions inside a disjunct also have a corresponding probability, represented as $(A \wedge B, p_1) \vee (C, p_2)$ for this example. Figure 15 presents the rules for generating derivation expressions for each node. Then, given the derivation expression E_n for node n, the

Algorithm 4 MakeDeltaPrecise(M, S_o, ϕ, δ)

Input: Maping M, set of output relations S_o, constraints ϕ, precision bound δ
Output: Mapping M that maps output facts to probabilities

1: $B \leftarrow$ BoundBounds(M, ϕ, S_o)
2: ▷ Compute upper and lower bounds for the bounds
3: **for** node $R \in S_o$ **do**
4: $(l^-, l^+, u^-, u^+) \leftarrow B[R]$
5: $(l^-, l^+) \leftarrow$ BinarySearch$(l^-, l^+, \phi, \delta, R, \text{true})$
6: $(u^-, u^+) \leftarrow$ BinarySearch$(u^-, u^+, \phi, \delta, R, \text{false})$
7: $M[R] \leftarrow (l^-, u^+)$
8: **return** M

Algorithm 5 BoundBounds(M, ϕ, S_o)

Input: Mapping M, constraints ϕ, set of output relations S_o
Output: A new mapping B, where $B[n]$ is a quadruple (l^-, l^+, u^-, u^+) where l^-, l^+ (resp. u^-, u^+) are lower and upper bounds for the ground truth lower (resp. upper) bound

1: **for** node $R \in S_o$ **do**
2: $(l, u) \leftarrow M[R]$
3: $(l^-, l^+) \leftarrow$ MakeSAT$(l, l, \phi, \texttt{Expr}(R), \text{true})$
4: $(u^-, u^+) \leftarrow$ MakeSAT$(u, u, \phi, \texttt{Expr}(R), \text{false})$
5: $B[R] \leftarrow (l^-, l^+, u^-, u^+)$
6: **return** B

ApproxExpr procedure (called at line 3 in Algorithm 3) computes the approximate upper and lower bounds for E_n using the rules presented in Figure 16. Given a derivation expression E, these rules derive judgments of the form $\Sigma, D \vdash E \rightsquigarrow [l, u]$ indicating that l, u are lower and upper bounds on the probability of expression E evaluating to true. To compute these lower and upper bounds, we leverage the results of the correlation type analysis (stored in Σ) as well as known statistical inequalities provided in Table 1 for different correlation types [45].

Theorem 4. *Let D be a* Praline *program with derivation graph G, and suppose Σ is a sound correlation environment for D. Also, let $M' =$ DeriveApproxBounds(G, D, Σ) and let $M =$ Solve(D) (Algorithm 1). For every output relation O of D such that $M(O) = (l^*, u^*)$ and $M'(O) = (l, u)$, we have $l \leq l^* \leq u^* \leq u$.*

7.3 Iterative Refinement of Probability Bounds

In this section, we describe the MakeDeltaPrecise algorithm that iteratively tightens the computed probability bounds until it is within some δ of the ground-truth. The key idea is to combine the algorithm from Section 6 with the approximate bounds as illustrated in Figure 7. In this Figure 7, l^* and u^* denote the ground truth (but unknown) probability bounds for relation R, and l and u denote the approximate probability bounds for R, computed as described in Section 7.2. Thus, it is always the case that $l \leq l^*$ and $u^* \leq u$. Our key observation is that the constraint ϕ generated in Section 6.3 partitions this space into three regions:

- **Region 1:** This is the region $\psi_1 = (l \leq \text{Expr}(R) < l^*)$, where $\text{Expr}(R)$ denotes the symbolic expression generated for R, as described in Section 6.4. Since the ground truth is $l^* \leq \text{Expr}(R) \leq u^*$, $\psi_1 \wedge \phi$ must be unsatisfiable.
- **Region 2:** This is the "ground truth" region $\psi_2 = l^* \leq \text{Expr}(R) \leq u^*$; thus, $\phi \wedge \psi_2$ is satisfiable.
- **Region 3:** This is the region $\psi_3 = (u^* < \text{Expr}(R) \leq u)$, so $\psi_3 \wedge \phi$ is again unsatisfiable.

As illustrated in Figure 7, the idea is to repeatedly increase l (resp. u) by ϵ until the formula $l_i \leq \text{Expr}(R) \leq l_i + \epsilon \wedge \phi$ (resp. $u_i - \epsilon \leq \text{Expr}(R) \leq u_i \wedge \phi$) becomes satisfiable. When this procedure terminates, we can obtain lower and upper bounds (l^-, l^+) for l^* as well as bounds (u^-, u^+) for u^*. We can then perform binary search until the distance between the two becomes less than δ.

This discussion is summarized in Algorithm 4. The MakeDeltaPrecise procedure first calls BoundBounds to compute upper and lower bounds for l^*, u^*, as depicted in Figure 7. As shown in Algorithm 5 (and its auxiliary procedure MakeSAT in Algorithm 6), BoundBounds increments (resp. decrements) l (resp. u) until we get into the SAT region in Figure 7. Upon termination of MakeSAT, (l^-, l^+) (resp. (u^-, u^+)) provide lower and upper bounds for l^* (resp. u^*). Then, for each relation R, MakeDeltaPrecise calls BinarySearch (Algorithm 7) to find a δ-optimal solution.

Algorithm 6 MakeSAT(l, u, ϕ, e, b)

Input: Lower bound l, upper bound u, constraints ϕ, expression e, boolean flag b. Boolean flag b indicating if we are dealing with a lower or upper bound
Output: Updated lower bound l and upper bound u

1: **while** UNSAT($\phi \wedge l \leq e \leq u$) **do**
2: $(l, u) \leftarrow$ b ? $(u, \text{Inc}(u))$
3: : $(\text{Dec}(l), l)$
4: **return** (l, u)

Algorithm 7 BinarySearch(l, u, ϕ, δ, R, b)

Input: Lower bound l, upper bound u, constraints ϕ, error bound δ, output relation R, boolean flag low
Output: Updated lower bound l and upper bound u

1: **while** $(u - l) \geq \delta$ **do**
2: $mid \leftarrow (l + u)/2$
3: $(l', u') \leftarrow \text{low?}(l, mid) : (mid, u)$
4: **if** UNSAT($\phi \wedge (l' \leq \text{Expr}(R) \leq u')$) **then**
5: $(l, u) \leftarrow \text{low?}(mid, u) : (l, mid)$
6: **else** $(l, u) \leftarrow \text{low?}(l, mid) : (mid, u)$
7: **return** (l, u)

When BinarySearch terminates, the returned interval $[l^-, l^+]$ is guaranteed to contain l^*, with $l^+ - l^- \leq \delta$. The same guarantee also applies to the upper bound.

Theorem 5. *Let (l^*, u^*) be the ground truth probability bounds for relation R, and let (l, u) be the bounds computed by* MakeDeltaPrecise. *Then, we have $l^* - \delta \leq l \leq l^*$ and $u^* \leq u \leq u^* + \delta$.*

8 Implementation

We have implemented our proposed approach in a tool called Praline written in C++. Praline instruments the solving procedure of Souffle [29] to generate the derivation graph and utilizes the Gurobi [10] solver for optimization and the Cvc5 [6] SMT solver for satisfiability.

Derivation graph generation. Datalog solvers such as Souffle avoid generating the same relation from different rules. For instance, if an output relation O has already been derived using a rule R, the Datalog solver would avoid applying other rules to derive O again. However, to compute the probability of O, we need all possible ways of deriving it; thus, our implementation modifies Souffle to generate the complete derivation graph. It also augments the derivation graph to keep track of rule probabilities.

Inference of correlation classes. While Praline allows the user to explicitly specify correlation classes (e.g., via the `corr` declaration) , it does not require them to do so. In particular, Praline constructs a dependency graph between input facts based on the specified conditional probabilities and assumes that a pair of input facts are in the same correlation class iff they belong to the same connected component. This default behavior can be overridden by users by explicitly specifying correlation classes.

Optimized satisfiability checks. Recall that the iterative refinement technique from Section 7.3 requires repeatedly checking satisfiability until the constraint becomes satisfiable. However, because the overwhelming majority of these calls return unsatisfiable, we simplify the problem by overapproximating the constraints until the overapproximation becomes satisfiable, in which case we switch to the exact encoding. The key idea underlying our over-approximation is as follows. While the exact encoding introduces joint probability variables over the input facts, we can instead introduce joint probability variables over *intermediate relations* that are k steps from the root node and encode the known correlations between them as part of the constraint. We provide an example of this encoding in Appendix A.4. k is not a fixed value; instead, k is selected dynamically. Details on how k is determined are provided in the Appendix.

Handling very large correlation classes. In cases where correlation classes become prohibitively large, even computing joint probability distributions within a *single* correlation class may be infeasible. For instance, some outliers in our experimental evaluation have hundreds of input facts in the same correlation class, making it infeasible to reason precisely about the joint probability for the entire class. In such cases, our implementation retains soundness but may compromise

δ-exactness. In particular, for correlation classes whose size exceeds a predefined threshold, we do not compute correlation types as described in Phase 1 of Section 7.1; however, we still perform the static analysis from Phase 2, assuming that input correlations within the same class are unknown. Second, when performing iterative refinement of the probability bounds, we approximate the satisfiability check as described in the previous paragraph and do *not* switch to the fully precise encoding. However, we emphasize that even this approximate solution for handling excessively large correlation classes utilizes the exact same machinery described in the rest of the paper.

9 Evaluation

In this section, we now describe the results for the evaluation that is designed to answer the following questions:

- **RQ1:** How accurate are the probability bounds inferred by Praline?
- **RQ2:** How efficient/scalable is our method in inferring the probability bounds?
- **RQ3:** How impactful are the key technical ingredients underlying Praline?

9.1 Application Domains and Benchmarks

We evaluate Praline on two different application domains spanning 30 benchmarks, summarized in Table 2. The first category, labeled **SC** (for Side Channel) in Table 2, corresponds to 19 Datalog-based program analyses for detecting power side-channel leaks. The second category, labeled **Bayes**, consists of 11 Bayesian networks sourced from the bnlearn repository. We provide more information about each of these application domains below.

Table 2. Benchmark statistics, CC denotes "correlation class".

		Min	Max	Avg	Med
SC	# nodes	56	200,164	43,366	9,093
	# edges	3	173,617	33,284	5,919
	CC size	8	731	133	31
Bayes	# nodes	10	223	79	60
	# edges	11	328	128	77
	CC size	2	13	5	4

Side channel benchmarks. Our first application domain is power side channel detection – a problem that has received significant attention in recent work [57, 58]. For this domain, input facts in the Datalog program are derived from the programs under analysis and include implementations of well-known cryptographic protocols such as AES, SHA3, and MAC-Keccak. The Datalog rules describing the side channel analysis are taken from [58]. Hence, each benchmark in the side channel category corresponds to the "cross product" of an existing side channel detector [58] and a real-world implementation of a cryptographic protocol. However, since the original Datalog-based analyzer only outputs a yes/no answer (indicating a potential power side channel vulnerability), we extend the analysis to quantify leakage severity. Our extension retains the existing Datalog rules and input facts as is, but augments them with probabilities as well as conditional dependencies/probabilities between input facts.

In more detail, we obtain the probabilities of the Datalog rules describing the program analysis using a similar methodology to what has been described in prior work on quantitative Datalog-based race detection [43]. This involves instrumenting the program to count rule firings and using these counts to estimate the probability of each rule applying in practice. To obtain the probabilities of input facts, we first note that while some input facts remain deterministic, others are probabilistic due to variations in register allocation algorithms and hardware architectures. For instance, the probability of register sharing is estimated using empirical data from profiling a code corpus. Similarly, certain input facts are probabilistic as they stem from pre-analysis [58, 61] that infers semantic data dependencies from syntactic ones. The probability of such dependencies is derived from prior empirical studies measuring how often syntactic dependencies translate into actual data dependencies in compiled programs [58]. To quantify conditional probabilities between input facts, we leverage empirical co-occurrence statistics from an existing code corpus [57]. We analyze execution traces to measure how often certain conditions –such as register sharing information

and key-related data dependencies – appear together. These conditional probabilities are essential for accurately assessing the severity of detected side channels, as some Datalog rules depend on both register-sharing behavior and secret-dependent data flows between variables. Consequently, the overall probability of data leakage must account for these correlations between (probabilistic) input facts rather than treating them as independent events.

Bayesian network benchmarks. Our second set of benchmarks, labeled **Bayes** in Table 2, consists of 11 discrete Bayesian networks sourced from the bnlearn repository [49], encompassing a range of network sizes, including *small*, *medium*, and *large* examples. We derive these benchmarks by preserving the original network structure, designating nodes without incoming edges as input facts and all other nodes as output facts. To introduce partially known statistical correlations between input facts, we inject dependencies that reflect realistic co-occurrence patterns observed in empirical data. For example, in medical networks, we introduce correlations between demographic and lifestyle factors, such as smoking and being Asian, while in weather models, we correlate atmospheric conditions like humidity and precipitation, which often vary together. These correlations are not deterministically defined, meaning that while input facts are not mutually independent, their exact joint distribution remains unknown. This approach enables us to evaluate inference under conditions where partial dependency information is available.

Benchmark statistics. Table 2 summarizes key statistics for both benchmark categories. **#nodes** and **#edges** represent the number of nodes and hyperedges in the derivation graph, while **CC size** denotes the size of the largest correlation class. We report the minimum, maximum, average, and median values across all benchmarks in each category. As shown in Table 2, the 19 side-channel benchmarks present a greater computational challenge than the 11 Bayesian network benchmarks from [49], highlighting the complexity of probabilistic inference in security applications. These two categories illustrate distinct but complementary use cases for Praline, both of which involve correlated inputs where exact dependencies are not fully known.

9.2 Experimental Methodology and Set-up

Existing probabilistic extensions of logic programming languages do not account for conditional dependencies between inputs. To assess whether ProbLog[16] could serve as a baseline, we attempted to encode input correlations[4] using its *evidence* predicate (see Appendix A.1 for details). However, ProbLog successfully terminated in only 17 of 30 benchmarks and failed to terminate on the remaining 13. Moreover, even when it did terminate, it produced unsound results due to its inability to faithfully represent Praline programs. This limitation stems from fundamental expressiveness gaps—accurately encoding a single Praline program in ProbLog would require generating an infinite number of distinct ProbLog programs (Appendix A.1).

Given these limitations, we evaluate Praline against the constrained optimization approach introduced in Section 6, which we use as the baseline to produce the ground truth. Our evaluation compares this baseline with the proposed δ-exact algorithm, which enhances scalability while preserving precise probability bounds. Throughout the remainder of this section, Praline refers to the δ-exact method from Section 7, and "Constrained Optimization" refers to the baseline.

All experiments were conducted on macOS Sonoma 14.4.1 with a 3-hour time limit and a memory cap of 16GB. In our evaluation, we set the δ parameter to 0.01 for the δ-exact algorithm, as it offered a practical balance between runtime and accuracy. In general, δ controls a clear trade-off: smaller values produce tighter probability bounds but incur longer runtimes due to additional refinement iterations, whereas larger values yield faster computations at the cost of looser bounds.

[4]For a fair comparison, we also constructed modified versions of our benchmarks where all input facts are treated as independent; in these cases, Praline and ProbLog produced identical results whenever ProbLog successfully terminated.

9.3 Accuracy Evaluation

In this section, we evaluate the accuracy of the probability bounds inferred by Praline in two ways. First, for those benchmarks where exact inference (Constrained Optimization) terminates, we compare the bounds computed by Praline against the ground truth. Second, because Praline may compromise δ-exactness for very large correlation classes (recall Section 8), we evaluate the percentage of output facts for which Praline guarantees δ-exact inference.

9.3.1 Comparison with Ground Truth. For 12 of the 30 benchmarks used in our evaluation (covering 3,179 queried facts), exact inference terminates within the 3-hour time limit, allowing us to evaluate Praline's results against the Ground Truth values. Table 3 presents the results of this evaluation, where the **LB Error** column reports the average, minimum, and maximum deviations in the lower bound, and the **UB Error** column provides the same for the upper bound. Specifically, **LB Error** corresponds to $l^* - l$, while **UB Error** denotes $u - u^*$, where (l, u) are the probability bounds computed by the δ-exact procedure and (l^*, u^*) are the exact values obtained via constrained optimization. Despite the substantial efficiency gains of the δ-exact method (evaluated more thoroughly in Section 9.5), its computed probability bounds remain highly precise. Across all 12 benchmarks, the average lower and upper bound errors are just **0.00168** and **0.00177**, respectively—well within the specified δ threshold of 0.01 (i.e., 1%). These results demonstrate that our approximate inference method provides an effective alternative to exact inference while maintaining near-optimal accuracy.

Table 3. Accuracy Comparison

	LB Error	UB Error
Average	0.00168	0.00177
Min	0.00000	0.00000
Max	0.00904	0.00935

9.3.2 Evaluation of δ-Exactness. As mentioned in Section 8, the implementation of Praline gives up on δ-exactness in some cases to scale to programs with very large correlation classes. In this section, we evaluate for what percent of output facts Praline can guarantee δ-exactness of inference. In particular, the result of inference for an output fact is guaranteed to be δ-exact if either (1) the length of the inferred interval is $\leq \delta$, or (2) the (final) satisfiability check in Section 7.3 uses the exact encoding over input facts, theoretically guaranteeing δ-exactness. Note that while these constitute sufficient conditions for δ-exact inference, they are not necessary conditions, meaning that the numbers reported here form a *lower bound* on the percentage of δ-exact results. The result of this evaluation is presented in Table 4, where #facts denotes the number of output facts queried by the program. *Overall, at least 73% of the queried output facts are guaranteed to be δ-exact.* As stated earlier, this number is merely a lower bound on the percentage of δ-exact results, owing to the simple reason that we do not have a scalable method of computing ground truths for the remaining facts.

Table 4. δ-exactness rate

	#facts	δ%
SC	96,393	73%
Bayes	663	100%
Overall	97,056	73%

Result for RQ1: For the benchmarks where ground truth bounds are available, Praline produces precise probability bounds, with average lower/upper bound errors of 0.1%. Over all 30 benchmarks comprising 97,056 queried facts, *at least* 73% are guaranteed to be δ-exact.

9.4 Inference Time Evaluation

To answer our second research question, we evaluate Praline's inference efficiency and scalability.

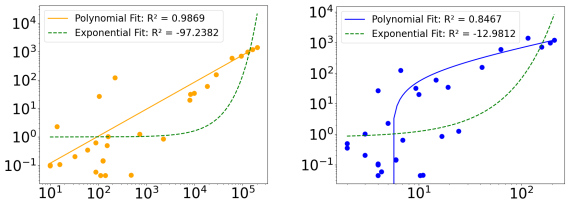
9.4.1 Inference Time. Table 5 shows the percentage of benchmarks that can be solved within a given time limit. As shown in this table, Praline is able to analyze 60% of the benchmarks in under 10 seconds and 76% in under 100 seconds. All 30 benchmarks terminate within the specified time limit, with the largest benchmark (with 200164

Table 5. Runtime

Time	Rate
< 1s	50%
1-10s	10%
10-100s	16%
100-1000s	17%
>1000s	7%

nodes, 173617 edges, 116 as the average correlation class size) taking 1414.23 seconds. This result shows that Praline is able to achieve practical inference times even for complex benchmarks.

9.4.2 Scalability Evaluation. To evaluate the scalability of Praline with respect to benchmark complexity, Figure 17 plots runtime against two complexity metrics. Figure 17a shows runtime (Y-axis, in seconds) versus the number of nodes (X-axis), while Figure 17b plots runtime against the average correlation class size. In Figure 17a, the orange curve (a *polynomial fit*) aligns closely with the data, achieving a high correlation coefficient of 0.9869, whereas the exponential fit (green dashed line) does not capture the trend well. Similarly, in Figure 17b, the blue curve (an *approximately polynomial fit*) provides a better fit than the exponential alternative.



(a) X = #Node (b) X = Average #Corr

Fig. 17. Runtime (Y-axis) vs. benchmark complexity (X-axis).

Result for RQ2: Praline is able to perform efficient probabilistic inference, with 76% of benchmarks being solved in under 100 seconds. Empirically, Praline scales polynomially with both the number of nodes and the average correlation class size.

9.5 Ablation Study

In this section, we describe a series of ablation studies designed to evaluate the impact of key ingredients of our approach. Specifically, we compare Praline against the following ablations:

- **Constrained optimization** This variant implements Algorithm 1.
- **Approx Only (AO)** This is a variant of Praline that computes loose approximate bounds using the approximated bound computation technique from Section 7.2. However, it does not utilize correlation types (correlation types are unknown), and it also does not perform refinement.
- **Approx+Correlation (AC)** This variant does not use iterative refinement for tightening the bound. That is, it computes approximate probabilities while leveraging correlation types.
- **Approx+Refinement (AR)** This ablation does not compute correlation types to assist the approximated bound computation. However, it does perform the refinement method of Section 7.3.

Among these ablations, we note that **AO** and **AC** do not provide any precision guarantees. Next, we evaluate the impact of each key ingredient on both inference time as well as accuracy.

9.5.1 Evaluation of Inference Time. Figure 18 explores the impact of various design choices on *inference time*. As we can see from Figure 18, the variants of **Praline** that do not have precision guarantees can perform inference more efficiently than all of the others. As expected, **Constrained Optimization** is the slowest and times out on the majority of the benchmarks. In contrast, both **Praline** and **AR** solve all benchmarks within the three-hour time limit, exhibiting similar performance in inference time. To see why **Praline** and **AR** have similar performance, note that computing correlation types adds some overhead, however, it also reduces time to perform iterative refinement, as the refinement algorithm starts with tighter bounds. Thus, overall, the computation of correlation types does not end up having a significant impact on inference time.

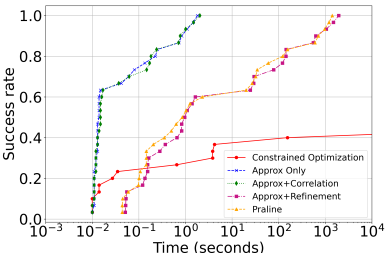


Fig. 18. Inference time.

9.5.2 Accuracy Evaluation. We now consider how the different variants of Praline perform in terms of accuracy. To quantify the impact of removing a specific feature in terms of accuracy, we consider a *bound tightness ratio (BTR)* metric, defined as follows:

$$\text{BTR}(\text{Praline}^{\Delta}) = \frac{u' - l'}{u - l}$$

Here, Praline^{Δ} refers to a specific ablation for Praline and $[l, u]$ and $[l', u']$ are the bounds produced by Praline and Praline^{Δ} respectively. Intuitively, the closer the BTR is to 1, the more accurate the ablation. The results of this evaluation are shown in Table 6. For each ablation (e.g., **AO**), we compute the BTR value for all output facts, and report the median, average, and maximum values. As is evident from the data, the intervals computed by Praline are much tighter compared to the other ablations, with **AO** producing the least precise intervals, followed by **AC**, and then **AR**.

Theoretically, **AR** and **Praline** should achieve similar accuracy, as both employ iterative refinement via binary search to tighten probability bounds. In practice, however, a key difference in how satisfiability checking is performed during refinement leads to **Praline** achieving at least twice the tightness of **AR**, as shown in Table 6.

Table 6. Ablation results of BTR

		Med	Avg	Max
BTR	**AO**	22.50	28.20	82.24
	AC	11.33	17.16	52.08
	AR	2.20	2.38	5.53

This accuracy gap stems from our implementation (Section 8). When correlation classes are large, the exact SAT encoding becomes computationally expensive and often intractable. To mitigate this, we introduce an optimized satisfiability checking strategy that leverages over-approximation. The precision of this optimization depends on the availability of accurate correlation type information. **Praline** explicitly infers correlation types in earlier stages (Section 7.1), enabling it to apply tighter over-approximations that closely match the exact encoding. In contrast, **AR** does not perform correlation type inference and conservatively treats all correlations as Unknown, resulting in looser encodings and reduced accuracy.

Result for RQ3: Praline strikes an effective balance between precision and inference time, delivering the most precise results across ablations that terminate on all benchmarks. Variants of Praline that do not perform refinement result in bounds that are $17 - 28\times$ worse on average.

10 Related Work

Probabilistic logic programming. Probabilistic programming allows programmers to model distributions and perform probabilistic sampling and inference, with systems like Pyro [9], Turing [23], Hakaru [41], SPPL [46], Dice [28] and PPL [51] leading the way. Recently, there has been significant progress in integrating logical reasoning into probabilistic programming to capture richer logical formalisms such as Horn clauses and first-order logic. Notable examples include probabilistic relational models [25], Markov logic networks [42], Bayesian logic programs [30], and probabilistic logic programming languages such as PRISM [48], LPADs [53], Blog [39], CP-logic[52], PPDL [5, 26], Datalogp [21], Scallop [37], and ProbLog [16, 17]. These formalisms extend existing logic programming languages like Prolog and Datalog by associating each rule with probabilities.

Among these languages, ProbLog [16, 17] and Scallop [37] focus on discrete distributions, which are closely related to our work. These techniques reduce probabilistic inference to weighted model counting (WMC) [54] and employ representations like binary decision diagrams (BDD) [12] to support efficient WMC. However, both ProbLog [16, 17] and Scallop [37] largely assume that input facts are independent and do not allow expressing general forms of conditional dependencies (other than providing the ability to express mutual exclusion between predicates). In contrast, Praline offers syntactic support to declare general forms of conditional dependency between input

facts, assign numerical probabilities to such dependencies, and assumes independence only in the absence of such declarations. To the best of our knowledge, JudgeD [55] is the only current Datalog extension that allows expressing dependencies between clauses by associating each input fact with a logical sentence. However, it neither supports negations nor does it allow specifying numerical probabilities for conditional dependencies.

Datalog for program analysis. Logic programming languages like Datalog have found numerous applications in program analysis, including for data-race detection [40, 43, 62], thread-modular analysis [34, 35], side-channel detection [57, 58], and points-to analysis [11, 38, 50, 60, 63]. Traditionally, Datalog-based analyses have been qualitative, but recent work [43, 63] has investigated quantitative analysis methods for inferring the likelihood of data races by incorporating probabilistic reasoning. These approaches, however, are constrained by their assumption about independence of input predicates, which our approach aims to address.

Exact probabilistic inference. Our method also relates to exact inference in graphical models like Bayesian networks and Markov networks. Exact inference techniques include weighted model counting [20, 28, 54], symbolic analysis [24], variable elimination [36], conjugacy [27], generating functions [31, 32], and optimization methods [4, 15]. However, applying these techniques directly to our problem domain is challenging for several reasons. Aside from the obvious structural differences between a Bayesian network and a Datalog derivation graph, our work also distinguishes itself from the Bayesian network setting by allowing for the specification of *incomplete* conditional dependencies for which it is not possible to compute a single probability value. Because of these important differences, prior approaches [28, 36] for speeding up probabilistic inference are unlikely to be effective in our setting.

Approximate probabilistic inference. Approximate inference techniques are primarily based on sampling methods [14, 33] such as Importance Sampling (IS), Markov Chain Monte Carlo (MCMC) and variational inference. However, these methods do not provide guarantees for results produced within a finite time frame. Some approaches [1, 8, 13, 19, 47] infer approximate posterior probabilities with guaranteed bounds; however, these methods typically focus on continuous rather than discrete distributions. Additionally, their guarantees rely on a countable set of sampled interval traces, which scale exponentially with the model's dimension [8].

Verifying probabilistic properties. There is large body of work on verifying probabilistic properties of programs, such as differential privacy and demographic fairness. For example, differential privacy can be expressed as relational properties of probabilistic computations involving expected values. Barthe et al. [7] propose a relational refinement type system and use approximate coupling to construct proofs. Albarghouthi and Hsu [2] and Wang et al. [59] simplify approximate coupling proofs to make it more automated. FairSquare [1], on the other hand, uses symbolic solving to verify if a program meets specified demographic fairness properties. While these approaches deal with probabilistic properties, they are largely orthogonal to our approach.

11 Conclusion

In this paper, we introduced a new probabilistic Datalog framework, Praline, which allows users to specify arbitrary statistical correlations between input facts, addressing a significant limitation in existing methods. Importantly, Praline is designed to handle scenarios where the statistical correlations between inputs are not fully known, allowing accurate probabilistic inference even under partial information. To solve this problem, we first proposed a constrained optimization approach that can compute exact probability bounds. Then, to address the scalability limitations of exact inference, we used this constrained optimization method in a more lightweight manner as

the basis of δ-exact algorithm that can approximate the true bounds, while guaranteeing that they are within distance δ of the ground truth. Our proposed δ-exact approach iteratively strengthens the approximated bounds through a synergistic combination of static analysis, approximation, and iterative refinement. Our empirical evaluation on 30 real-world probabilistic Datalog programs demonstrates that Praline can compute precise probability bounds, while scaling to large benchmarks with more than 200,000 relations. In contrast, the ablations of Praline that are not δ-exact infer significantly less accurate probability bounds, while exact inference does not scale. These experiments demonstrate that Praline strikes an effective balance between precision and inference time.

12 Data-Availability Statement

An artifact supporting the results of this paper is available on Zenodo [56]. Our tool depends on Datalog, SMT, and optimization solvers, which users will need to install separately. One requirement is a free academic license for Gurobi optimization solver, which can be easily obtained using an institutional email address.

Acknowledgments

We thank the anonymous reviewers for their constructive feedback. This work was conducted in a research group supported by NSF awards CCF-1762299, CCF-1918889, CNS-1908304, CCF-1901376, CNS-2120696, CCF- 2210831, and CCF-2319471, CCF-2422130, CCF-2403211 as well as a DARPA award under agreement HR00112590133.

References

[1] Aws Albarghouthi, Loris D'Antoni, Samuel Drews, and Aditya V Nori. 2017. Fairsquare: probabilistic verification of program fairness. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–30. https://doi.org/doi/10.1145/3133904

[2] Aws Albarghouthi and Justin Hsu. 2017. Synthesizing coupling proofs of differential privacy. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–30. https://doi.org/doi/10.1145/3158146

[3] V Aho Alfred, S Lam Monica, and D Ullman Jeffrey. 2007. *Compilers principles, techniques & tools.* pearson Education.

[4] Stephen H Bach, Matthias Broecheler, Bert Huang, and Lise Getoor. 2017. Hinge-loss markov random fields and probabilistic soft logic. *Journal of Machine Learning Research* 18, 109 (2017), 1–67.

[5] Vince Bárány, Balder Ten Cate, Benny Kimelfeld, Dan Olteanu, and Zografoula Vagena. 2017. Declarative probabilistic programming with datalog. *ACM Transactions on Database Systems (TODS)* 42, 4 (2017), 1–35. https://doi.org/10.1145/3132700

[6] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, et al. 2022. cvc5: A versatile and industrial-strength SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 415–442.

[7] Gilles Barthe, Marco Gaboardi, Emilio Jesús Gallego Arias, Justin Hsu, Aaron Roth, and Pierre-Yves Strub. 2015. Higher-order approximate relational refinement types for mechanism design and differential privacy. *ACM SIGPLAN Notices* 50, 1 (2015), 55–68. https://doi.org/doi/10.1145/2676726.2677000

[8] Raven Beutner, C-H Luke Ong, and Fabian Zaiser. 2022. Guaranteed bounds for posterior inference in universal probabilistic programming. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 536–551. https://doi.org/doi/10.1145/3519939.3523721

[9] Eli Bingham, Jonathan P Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D Goodman. 2019. Pyro: Deep universal probabilistic programming. *Journal of machine learning research* 20, 28 (2019), 1–6. https://doi.org/doi/abs/10.5555/3322706.3322734

[10] Bob Bixby. 2007. The gurobi optimizer. *Transp. Re-search Part B* 41, 2 (2007), 159–178.

[11] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*. 243–262. https://doi.org/doi/10.1145/1640089.1640108

[12] Randal E Bryant. 1986. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on* 100, 8 (1986), 677–691.

[13] Patrick Cousot and Michael Monerau. 2012. Probabilistic abstract interpretation. In *European Symposium on Programming*. Springer, 169–193.

[14] Adnan Darwiche. 2009. *Modeling and reasoning with Bayesian networks*. Cambridge university press.

[15] Erik Daxberger, Anastasia Makarova, Matteo Turchetta, and Andreas Krause. 2021. Mixed-variable Bayesian optimization. In *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*. 2633–2639.

[16] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. 2007. ProbLog: A probabilistic Prolog and its application in link discovery. In *IJCAI 2007, Proceedings of the 20th international joint conference on artificial intelligence*. IJCAI-INT JOINT CONF ARTIF INTELL, 2462–2467.

[17] Anton Dries, Angelika Kimmig, Wannes Meert, Joris Renkens, Guy Van den Broeck, Jonas Vlasselaer, and Luc De Raedt. 2015. Problog2: Probabilistic logic programming. In *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2015, Porto, Portugal, September 7-11, 2015, Proceedings, Part III 15*. Springer, 312–315.

[18] Hassan Eldib and Chao Wang. 2014. Synthesis of masking countermeasures against side channel attacks. In *Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings 26*. Springer, 114–130.

[19] Shenghua Feng, Mingshuai Chen, Han Su, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Naijun Zhan. 2023. Lower bounds for possibly divergent probabilistic programs. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (2023), 696–726. https://doi.org/doi/10.1145/3586051

[20] Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens, and Luc De Raedt. 2015. Inference and learning in probabilistic logic programs using weighted boolean formulas. *Theory and Practice of Logic Programming* 15, 3 (2015), 358–401. https://doi.org/abs/1304.6810

[21] Norbert Fuhr. 1995. Probabilistic datalog—a logic for powerful retrieval methods. In *Proceedings of the 18th annual international ACM SIGIR conference on Research and development in information retrieval*. 282–290. https://doi.org/doi/10.1145/215206.215372

[22] Sicun Gao, Jeremy Avigad, and Edmund M Clarke. 2012. δ-complete decision procedures for satisfiability over the reals. In *International Joint Conference on Automated Reasoning*. Springer, 286–300. https://doi.org/abs/1204.3513

[23] Hong Ge, Kai Xu, and Zoubin Ghahramani. 2018. Turing: a language for flexible probabilistic inference. In *International conference on artificial intelligence and statistics*. PMLR, 1682–1690.

[24] Timon Gehr, Sasa Misailovic, and Martin Vechev. 2016. PSI: Exact symbolic inference for probabilistic programs. In *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I 28*. Springer, 62–83.

[25] Lise Getoor, Nir Friedman, Daphne Koller, Avi Pfeffer, and Ben Taskar. 2007. Probabilistic relational models. (2007).

[26] Martin Grohe, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Peter Lindner. 2022. Generative datalog with continuous distributions. *J. ACM* 69, 6 (2022), 1–52. https://doi.org/doi/10.1145/3559102

[27] Matthew D Hoffman, Matthew J Johnson, and Dustin Tran. 2018. Autoconj: recognizing and exploiting conjugacy without a domain-specific language. *Advances in Neural Information Processing Systems* 31 (2018). https://doi.org/doi/10.5555/3327546.3327731

[28] Steven Holtzen, Guy Van den Broeck, and Todd Millstein. 2020. Scaling exact inference for discrete probabilistic programs. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–31. https://doi.org/doi/10.1145/3428208

[29] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. Soufflé: On synthesis of program analyzers. In *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II 28*. Springer, 422–430.

[30] Kristian Kersting and Luc De Raedt. 2007. Bayesian logic programming: Theory and tool. (2007).

[31] Lutz Klinkenberg, Christian Blumenthal, Mingshuai Chen, Darion Haase, and Joost-Pieter Katoen. 2024. Exact Bayesian Inference for Loopy Probabilistic Programs using Generating Functions. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (2024), 923–953. https://doi.org/doi/10.1145/3649844

[32] Lutz Klinkenberg, Tobias Winkler, Mingshuai Chen, and Joost-Pieter Katoen. 2023. Exact probabilistic inference using generating functions. *arXiv preprint arXiv:2302.00513* (2023).

[33] Daphne Koller and Nir Friedman. 2009. *Probabilistic graphical models: principles and techniques*. MIT press. https://doi.org/doi/10.5555/1795555

[34] Markus Kusano and Chao Wang. 2016. Flow-sensitive composition of thread-modular abstract interpretation. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 799–809. https://doi.org/doi/10.1145/2950290.2950291

[35] Markus Kusano and Chao Wang. 2017. Thread-modular static analysis for relaxed memory models. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 337–348. https://doi.org/doi/10.1145/3106237.3106243

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

[37] Ziyang Li, Jiani Huang, and Mayur Naik. 2023. Scallop: A language for neurosymbolic programming. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1463–1487. https://doi.org/doi/10.1145/3591280

[38] Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. 2016. From datalog to flix: A declarative language for fixed points on lattices. *ACM SIGPLAN Notices* 51, 6 (2016), 194–208. https://doi.org/doi/10.1145/2980983.2908096

[39] Brian Milch, Bhaskara Marthi, Stuart Russell, David Sontag, Daniel L Ong, and Andrey Kolobov. 2007. BLOG: Probabilistic models with unknown objects. (2007).

[40] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective static race detection for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 308–319. https://doi.org/doi/10.1145/1133981.1134018

[41] Praveen Narayanan, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov. 2016. Probabilistic inference by program transformation in Hakaru (system description). In *Functional and Logic Programming: 13th International Symposium, FLOPS 2016, Kochi, Japan, March 4-6, 2016, Proceedings 13*. Springer, 62–79.

[42] Feng Niu, Christopher Ré, AnHai Doan, and Jude Shavlik. 2011. Tuffy: Scaling up Statistical Inference in Markov Logic Networks using an RDBMS. *Proceedings of the VLDB Endowment* 4, 6 (2011). https://doi.org/doi/10.14778/1978665.1978669

[43] Mukund Raghothaman, Sulekha Kulkarni, Kihong Heo, and Mayur Naik. 2018. User-guided program reasoning using Bayesian inference. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 722–735. https://doi.org/doi/10.1145/3296979.3192417

[44] Thomas W Reps. 1995. Demand interprocedural program analysis using logic databases. In *Applications of Logic Databases*. Springer, 163–196.

[45] Ludger Rüschendorf. [n. d.]. Fréchet-bounds and their applications. In *Advances in Probability Distributions with Given Marginals: beyond the copulas*. Springer, 151–187.

[46] 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*. 804–819. https://doi.org/doi/10.1145/3453483.3454078

[47] Sriram Sankaranarayanan, Aleksandar Chakarov, and Sumit Gulwani. 2013. Static analysis for probabilistic programs: inferring whole program properties from finitely many paths. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. 447–458.

[48] Taisuke Sato. 1995. A statistical learning method for logic programs with distribution semantics. (1995).

[49] Marco Scutari. 2009. Learning Bayesian networks with the bnlearn R package. *arXiv preprint arXiv:0908.3817* (2009).

[50] Yannis Smaragdakis, George Kastrinis, and George Balatsouras. 2014. Introspective analysis: context-sensitivity, across the board. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 485–495. https://doi.org/doi/10.1145/2499370.2462179

[51] Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. 2018. An introduction to probabilistic programming. *arXiv preprint arXiv:1809.10756* (2018).

[52] Joost Vennekens, Marc Denecker, and Maurice Bruynooghe. 2009. CP-logic: A language of causal probabilistic events and its relation to logic programming. *Theory and practice of logic programming* 9, 3 (2009), 245–308. https://doi.org/10.1017/S1471068409003767

[53] Joost Vennekens, Sofie Verbaeten, and Maurice Bruynooghe. 2004. Logic programs with annotated disjunctions. In *Logic Programming: 20th International Conference, ICLP 2004, Saint-Malo, France, September 6-10, 2004. Proceedings 20*. Springer, 431–445.

[54] Jonas Vlasselaer, Guy Van den Broeck, Angelika Kimmig, Wannes Meert, and Luc De Raedt. 2016. Tp-compilation for inference in probabilistic logic programs. *International Journal of Approximate Reasoning* 78 (2016), 15–32. https://doi.org/10.1016/j.ijar.2016.06.009

[55] Brend Wanders, Maurice van Keulen, and Jan Flokstra. 2016. Judged: a probabilistic datalog with dependencies. In *Workshops at the Thirtieth AAAI Conference on Artificial Intelligence*.

[56] Jingbo Wang, Shashin Halalingaiah, Weiyi Chen, Chao Wang, and Isil Dillig. 2025. Reproduction Package for Article 'Probabilistic Inference for Datalog with Correlated Inputs'. ACM. https://doi.org/10.5281/zenodo.15760564

[57] Jingbo Wang, Chungha Sung, Mukund Raghothaman, and Chao Wang. 2021. Data-driven synthesis of provably sound side channel analyses. In 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). https://doi.org/doi/10.1109/ICSE43902.2021.00079

[58] Jingbo Wang, Chungha Sung, and Chao Wang. 2019. Mitigating power side channels during compilation. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 590–601. https://doi.org/doi/10.1145/3338906.3338913

[59] Yuxin Wang, Zeyu Ding, Guanhong Wang, Daniel Kifer, and Danfeng Zhang. 2019. Proving differential privacy with shadow execution. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 655–669. https://doi.org/doi/10.1145/3314221.3314619

[60] John Whaley, Dzintars Avots, Michael Carbin, and Monica S Lam. 2005. Using Datalog with binary decision diagrams for program analysis. In *Asian Symposium on Programming Languages and Systems*. Springer, 97–118. https://doi.org/doi/10.1007/11575467_8

[61] Jun Zhang, Pengfei Gao, Fu Song, and Chao Wang. 2018. SCInfer: Refinement-based verification of software countermeasures against side-channel attacks. In *International Conference on Computer Aided Verification*. Springer, 157–177.

[62] Xin Zhang, Radu Grigore, Xujie Si, and Mayur Naik. 2017. Effective interactive resolution of static analysis alarms. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–30. https://doi.org/doi/10.1145/3133881

[63] Xin Zhang, Ravi Mangal, Radu Grigore, Mayur Naik, and Hongseok Yang. 2014. On abstraction refinement for program analyses in Datalog. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 239–248. https://doi.org/doi/10.1145/2666356.2594327

Received 2025-03-25; accepted 2025-08-12