Declarative Probabilistic Programming with Datalog

Vince Barany[1], Balder ten Cate[1], Benny Kimelfeld[2], Dan Olteanu[3], and Zografoula Vagena[4]

1 LogicBlox*
2 Technion, Israel and LogicBlox
3 University of Oxford, UK and LogicBlox
4 LogicBlox

Abstract

Probabilistic programming languages are used for developing statistical models, and they typically consist of two components: a specification of a stochastic process (the prior), and a specification of observations that restrict the probability space to a conditional subspace (the posterior). Use cases of such formalisms include the development of algorithms in machine learning and artificial intelligence. We propose and investigate an extension of Datalog for specifying statistical models, and establish a declarative probabilistic-programming paradigm over databases. Our proposed extension provides convenient mechanisms to include common numerical probability functions; in particular, conclusions of rules may contain values drawn from such functions. The semantics of a program is a probability distribution over the possible outcomes of the input database with respect to the program. Observations are naturally incorporated by means of integrity constraints over the extensional and intensional relations. The resulting semantics is robust under different chases and invariant to rewritings that preserve logical equivalence.

1998 ACM Subject Classification D.1.6 Logic Programming; G.3 Probability and Statistics; H.2.3 Languages

Keywords and phrases Chase, Datalog, probability measure space, probabilistic programming

Digital Object Identifier 10.4230/LIPIcs.ICDT.2016.1

1 Introduction

Languages for specifying general statistical models are commonly used in the development of machine learning and artificial intelligence algorithms for tasks that involve inference under uncertainty. A substantial effort has been made on developing such formalisms and corresponding system implementations. An actively studied concept in that area is that of *Probabilistic Programming* (PP) [20], where the idea is that the programming language allows for specifying general random procedures, while the system *executes* the program not in the standard programming sense, but rather by means of *inference*. Hence, a PP system is built around a language and an (approximate) inference engine, which typically makes use of Markov Chain Monte Carlo methods (e.g., the Metropolis-Hastings algorithm). The relevant inference tasks can be viewed as probability-aware aggregate operations over all possible worlds, that is, possible outcomes of the program. Examples of such tasks include finding the most likely possible world, or estimating the probability of an event. Recently,

* Now at Google, Inc.

DARPA initiated the project *Probabilistic Programming for Advancing Machine Learning (PPAML)*, aimed at advancing PP systems (with a focus on a specific collection of systems, e.g., [30, 32, 40]) towards facilitating the development of algorithms and software that are based on machine learning.

In probabilistic programming, a statistical model is typically phrased by means of two components. The first component is a *generative process* that produces a random possible world by straightforwardly following instructions with randomness, and in particular, sampling from common numerical probability functions; this gives the *prior distribution*. The second component allows to phrase constraints that the relevant possible worlds should satisfy, and, semantically, transforms the prior distribution into the *posterior distribution*—the subspace obtained by conditioning on the constraints.

As an example, in *supervised text classification* (e.g., spam detection) the goal is to classify a text document into one of several known classes (e.g., spam/non-spam). Training data consists of a collection of documents labeled with classes, and the goal of learning is to build a model for predicting the classes of unseen documents. One common approach to this task assumes a generative process that produces random *parameters* for every class, and then uses these parameters to define a generator of random words in documents of the corresponding class [31, 33]. The prior distribution thus generates parameters and documents for each class, and the posterior is defined by the actual documents of the training data. In *unsupervised* text classification the goal is to cluster a given set of documents, so that different clusters correspond to different topics (not known in advance). Latent Dirichlet Allocation [10] approaches this problem in a similar generative way as the above, with the addition that each document is associated with a distribution over topics.

A Datalog program is a set of logical rules, interpreted in the context of a relational database (where database relations are also called the *extensional relations*), that are used to define additional relations (known as the *intensional relations*). Datalog has traditionally been used as a database query language. In recent years, however, it has found new applications in data integration, information extraction, networking, program analysis, security, cloud computing, and enterprise software development [23]. In each of these applications, being declarative, Datalog makes specifications easier to write (sometimes with orders-of-magnitude fewer lines of code than imperative code, e.g., [28]), and to comprehend and maintain.

In this work, we extend Datalog with the ability to program statistical models. In par with existing languages for PP, our proposed extension consists of two parts: a *generative Datalog program* that specifies a prior probability space over (finite or infinite) sets of facts that we call *possible outcomes*, and a definition of the posterior probability by means of *observations*, which come in the form of ordinary logical constraints over the extensional and intensional relations. We subscribe to the premise of the PP community (and PPAML in particular) that this paradigm has the potential of substantially facilitating the development of applications that involve machine learning for inferring missing or uncertain information. Indeed, probabilistic variants are explored for the major programming languages, such as C [37], Java [27], Scala [40], Scheme [30] and Python [38] (we discuss the relationship of this work to related literature in Section 6). At LogicBlox, we are interested in extending our Datalog-based LogiQL [22] with PP to enable and facilitate the development of predictive analysis [6]. We believe that, once the semantics becomes clear, Datalog can offer a natural and appealing basis for PP, since it has an inherent (and well studied) separation between given data (EDB), generated data (IDB), and conditioning (constraints).

The main challenge, when attempting to extend Datalog with probabilistic programming constructs, is to retain the inherent features of Datalog. Specifically, the semantics of Datalog does not depend on the order by which the rules are resolved (chased). Hence, it is safe to provide a Datalog engine with the ability to decide on the chasing order that is estimated to be more efficient. Another feature is invariance under logical equivalence: two Datalog programs have the same semantics whenever their rules are equivalent when viewed as theories in first-order logic. Hence, it is safe for a Datalog engine to rewrite a program, as long as logical equivalence is preserved.

For example, consider an application where we want to predict the number of visits of clients to some local service (e.g., a doctor's office). For simplicity, suppose that we have a schema with the following relations: $\mathrm{LivesIn}(\mathrm{person}, \mathrm{city})$, $\mathrm{WorksIn}(\mathrm{person}, \mathrm{employer})$, $\mathrm{LocatedIn}(\mathrm{company}, \mathrm{city})$, and $\mathrm{AvgVisits}(\mathrm{city}, \mathrm{avg})$. The following rule provides an appealing way to model the generation of a random number of visits for a person.

$$\mathrm{Visits}(p, \mathsf{Poisson}[\lambda]) \leftarrow \mathrm{LivesIn}(p, c), \mathrm{AvgVisits}(c, \lambda) \tag{1}$$

The conclusion of this rule involves sampling values from a parameterized probability distribution. Next, suppose that we do not have all the addresses of persons, and we wish to expand the simulation with employer cities. Then we might use the following additional rule.

$$\mathrm{Visits}(p, \mathsf{Poisson}[\lambda]) \leftarrow \mathrm{WorksIn}(p, e), \mathrm{LocatedIn}(e, c), \mathrm{AvgVisits}(c, \lambda) \tag{2}$$

Now, it is not clear how to interpret the semantics of Rules (1) and (2) in a manner that retains the declarative nature of Datalog. If, for a person p, the right sides of both rules are true, should both rules "fire" (i.e., should we sample the Poisson distribution twice)? And if p works in more than one company, should we have one sample per company? And if p lives in one city but works in another, which rule should fire? If only one rule fires, then the semantics becomes dependent on the chase order. To answer these questions, we need to properly define what it means for the head of a rule to be *satisfied* when it involves randomness such as $\mathsf{Poisson}[\lambda]$.

Furthermore, consider the following (standard) rewriting of the above program.

$$\begin{aligned}
\mathrm{PersonCity}(p, c) &\leftarrow \mathrm{LivesIn}(p, c)\\
\mathrm{PersonCity}(p, c) &\leftarrow \mathrm{WorksIn}(p, e), \mathrm{LocatedIn}(e, c)\\
\mathrm{Visits}(p, \mathsf{Poisson}[\lambda]) &\leftarrow \mathrm{PersonCity}(p, c), \mathrm{AvgVisits}(c, \lambda)
\end{aligned}$$

As a conjunction of first-order sentences, the rewritten program is equivalent to the previous one; we would therefore like the two programs to have the same semantics. In rule-based languages with a factor-based semantics, such as *Markov Logic Networks* [15] or *Probabilistic Soft Logic* [11], the above rewriting may change the semantics dramatically.

We introduce *PPDL*, a purely declarative probabilistic programming language based on Datalog. The generative component of a PPDL program consists of rules extended with constructs to refer to conventional parameterized numerical probability functions (e.g., Poisson, geometrical, etc.). Specifically, these mechanisms allow sampling values from the given parameterized distributions in the conclusion of a rule (and if desired, use these values as parameters of other distributions). In this paper, our focus is on discrete numerical distributions (the framework we introduce admits a natural generalization to continuous distributions, such as Gaussian or Pareto, but we defer the details of this to future work). Semantically, a PPDL program associates to each input instance I a probability distribution over *possible outcomes*. In the case where all the possible outcomes are finite, we get a discrete probability distribution, and the probability of a possible outcome can be defined immediately from its content. But in general, a possible outcome can be infinite, and moreover, the set of all possible outcomes can be uncountable. Hence, in the general case we obtain a probability measure space. We define a natural notion of a *probabilistic chase* where existential variables are produced by invoking the corresponding numerical distributions. We define a measure space based on a chase, and prove that this definition is robust, in the sense that the same probability measure is obtained no matter which chase order is used.

A short version of this paper has appeared in the 2015 Alberto Mendelzon International Workshop [46]. Due to lack of space, proofs will appear in the full version of this paper.

2 Preliminaries

In this section we give basic notation and definitions that we use throughout the paper.

Schemas and instances. A (*relational*) *schema* is a collection $\mathcal{S}$ of *relation symbols*, where each relation symbol R is associated with an *arity*, denoted $\mathsf{arity}(R)$, which is a natural number. An *attribute* of a relation symbol R is any number in $\{1, \dots, \mathsf{arity}(R)\}$. For simplicity, we consider here only databases over real numbers; our examples may involve strings, which we assume are translatable into real numbers. A *fact* over a schema $\mathcal{S}$ is an expression of the form $R(c_1, \dots, c_n)$ where R is an n-ary relation in $\mathcal{S}$ and $c_1, \dots, c_n \in \mathbb{R}$. An *instance* I over $\mathcal{S}$ is a finite set of facts over $\mathcal{S}$. We denote by R^I the set of all tuples $(c_1, \dots, c_n)$ such that $R(c_1, \dots, c_n) \in I$.

Datalog programs. PPDL extends Datalog without the use of existential quantifiers. However, we will make use of existential rules indirectly in the definition of the semantics. For this reason, we review here Datalog as well as existential Datalog. Formally, an *existential Datalog program*, or Datalog$^\exists$ *program*, is a triple $\mathcal{D} = (\mathcal{E}, \mathcal{I}, \Theta)$ where: (1) $\mathcal{E}$ is a schema, called the *extensional database* (EDB) schema, (2) $\mathcal{I}$ is a schema, called the *intensional database* (IDB) schema, disjoint from $\mathcal{E}$, and (3) Θ is a finite set of Datalog$^\exists$ *rules*, that is, first-order formulas of the form $\forall \mathbf{x}\big[\, (\exists \mathbf{y} \psi(\mathbf{x}, \mathbf{y})) \leftarrow \varphi(\mathbf{x}) \,\big]$ where $\varphi(\mathbf{x})$ is a conjunction of atomic formulas over $\mathcal{E} \cup \mathcal{I}$ and $\psi(\mathbf{x}, \mathbf{y})$ is an atomic formula over $\mathcal{I}$, such that each variable in $\mathbf{x}$ occurs in φ. Here, by an *atomic formula* (or, *atom*) we mean an expression of the form $R(t_1, \dots, t_n)$ where R is an n-ary relation and $t_1, \dots, t_n$ are either constants (i.e., numbers) or variables. For readability's sake, we omit the universal quantifier and the parentheses around the conclusion (left-hand side), and write simply $\exists \mathbf{y} \psi(\mathbf{x}, \mathbf{y}) \leftarrow \varphi(\mathbf{x})$. *Datalog* is the fragment of Datalog$^\exists$ where the conclusion of each rule is an atomic formula without existential quantifiers.

Let $\mathcal{D} = (\mathcal{E}, \mathcal{I}, \Theta)$ be a Datalog$^\exists$ program. An *input instance* for $\mathcal{D}$ is an instance I over $\mathcal{E}$. A *solution* of I w.r.t. $\mathcal{D}$ is a possibly infinite set F of facts over $\mathcal{E} \cup \mathcal{I}$, such that $I \subseteq F$ and F satisfies all rules in Θ (viewed as first-order sentences). A *minimal solution* of I (w.r.t. $\mathcal{D}$) is a solution F of I such that no proper subset of F is a solution of I. The set of all, finite and infinite, minimal solutions of I w.r.t. $\mathcal{D}$ is denoted by $\mathsf{min\text{-}sol}_{\mathcal{D}}(I)$, and the set of all *finite* minimal solutions is denoted by $\mathsf{min\text{-}sol}^{\mathsf{fin}}_{\mathcal{D}}(I)$. It is a well known fact that, if $\mathcal{D}$ is a *Datalog program* (that is, without existential quantifiers), then every input instance I has a unique minimal solution, which is finite, and therefore $\mathsf{min\text{-}sol}^{\mathsf{fin}}_{\mathcal{D}}(I) = \mathsf{min\text{-}sol}_{\mathcal{D}}(I)$.

Probability spaces. We separately consider *discrete* and *continuous* probability spaces. We initially focus on the discrete case; there, a *probability space* is a pair (Ω, π), where Ω is a finite or countably infinite set, called the *sample space*, and $\pi : \Omega \to [0, 1]$ is such that $\sum_{o \in \Omega} \pi(o) = 1$. If (Ω, π) is a probability space, then π is a *probability distribution* over Ω. We say that π is a *numerical* probability distribution if $\Omega \subseteq \mathbb{R}$. In this work we focus on discrete numerical distributions.

A *parameterized* probability distribution is a function $\delta : \Omega \times \mathbb{R}^k \to [0, 1]$, such that

$\delta(\cdot, \mathbf{p}) : \Omega \to [0,1]$ is a probability distribution for all $\mathbf{p} \in \mathbb{R}^k$. We use $\mathsf{pardim}(\delta)$ to denote the parameter dimension k. For presentation's sake, we may write $\delta(o|\mathbf{p})$ instead of $\delta(o, \mathbf{p})$. Moreover, we denote the (non-parameterized) distribution $\delta(\cdot|\mathbf{p})$ by $\delta[\mathbf{p}]$. An example of a parameterized distribution is $\mathsf{Flip}(\cdot|p)$, where Ω is $\{0,1\}$, and for a parameter $p \in [0,1]$ we have $\mathsf{Flip}(1|p) = p$ and $\mathsf{Flip}(0|p) = 1 - p$. Another example is $\mathsf{Poisson}(\cdot|\lambda)$, where $\Omega = \mathbb{N}$, and for a parameter $\lambda \in (0, \infty)$ we have $\mathsf{Poisson}(x|\lambda) = \lambda^x e^{-\lambda}/x!$. In Section 7 we discuss the extension of our framework to models that have a variable number of parameters, and to continuous distributions.

Let Ω be a set. A *σ-algebra* over Ω is a collection $\mathcal{F}$ of subsets of Ω, such that $\mathcal{F}$ contains Ω and is closed under complement and countable unions. (Implied properties include that $\mathcal{F}$ contains the empty set, and that $\mathcal{F}$ is closed under countable intersections.) If $\mathcal{F}'$ is a nonempty collection of subsets of Ω, then the closure of $\mathcal{F}'$ under complement and countable unions is a σ-algebra, and it is said to be *generated* by $\mathcal{F}'$. A *probability measure space* is a triple $(\Omega, \mathcal{F}, \pi)$, where: *(1)* Ω is a set, called the *sample space*, *(2)* $\mathcal{F}$ is a σ-algebra over Ω, *and (3)* $\pi : \mathcal{F} \to [0,1]$, called a *probability measure*, is such that $\pi(\Omega) = 1$, and $\pi(\cup\mathcal{E}) = \sum_{e \in \mathcal{E}} \pi(e)$ for every countable set $\mathcal{E}$ of pairwise-disjoint elements of $\mathcal{F}$.

3 Generative Datalog

A Datalog program without existential quantifiers specifies how to obtain a minimal solution from an input instance by producing the set of inferred IDB facts. In this section we present *generative Datalog programs*, which specify how to infer a *distribution over possible outcomes* given an input instance. In Section 5 we will complement generative programs with *constraints* to establish the PPDL framework.

3.1 Syntax

The syntax of a generative Datalog program is defined as follows.

▶ **Definition 1** (GDatalog[Δ]). Let Δ be a finite set of parameterized numerical distributions.

1. A Δ-*term* is a term of the form $\delta[\![p_1, \ldots, p_k]\!]$ where $\delta \in \Delta$ is a parameterized distribution with $\mathsf{pardim}(\delta) = \ell \le k$, and each p_i $(i = 1, \ldots, k)$ is a variable or a constant. To improve readability, we will use a semicolon to separate the first ℓ arguments (corresponding to the distribution parameters) from the optional other arguments (which we will call the *event signature*), as in $\delta[\![\mathbf{p}; \mathbf{q}]\!]$. When the event signature is empty (i.e., when $k = \ell$), we write $\delta[\![\mathbf{p};]\!]$.[1]
2. A Δ-*atom* in a schema $\mathcal{S}$ is an atomic formula $R(t_1, \ldots, t_n)$ with $R \in \mathcal{S}$ an n-ary relation, such that exactly one term t_i $(i = 1, \ldots, n)$ is a Δ-term and the other terms t_j are variables and/or constants.[2]
3. A *GDatalog*[Δ] *rule* over a pair of disjoint schemas $\mathcal{E}$ and $\mathcal{I}$ is a first-order sentence of the form $\forall \mathbf{x}(\psi(\mathbf{x}) \leftarrow \phi(\mathbf{x}))$ where $\phi(\mathbf{x})$ is a conjunction of atoms in $\mathcal{E} \cup \mathcal{I}$ and $\psi(\mathbf{x})$ is either an atom in $\mathcal{I}$ or a Δ-atom in $\mathcal{I}$.
4. A *GDatalog*[Δ] *program* is a triple $\mathcal{G} = (\mathcal{E}, \mathcal{I}, \Theta)$, where $\mathcal{E}$ and $\mathcal{I}$ are disjoint schemas and Θ is a finite set of GDatalog[Δ] rules over $\mathcal{E}$ and $\mathcal{I}$.

[1] Intuitively, $\delta[\![\mathbf{p}; \mathbf{q}]\!]$ denotes a sample from the distribution $\delta(\cdot|\mathbf{p})$ where different samples are drawn for different values of the event signature $\mathbf{q}$ (cf. Example 2).
[2] The restriction to at most one Δ-term per atom is only for presentational purposes, cf Section 3.5.

House	
id	*city*
NP1	Napa
NP2	Napa
YC1	Yucaipa

Business	
id	*city*
NP3	Napa
YC1	Yucaipa

City	
name	*burglaryrate*
Napa	0.03
Yucaipa	0.01

AlarmOn
unit
NP1
YC1
YC2

Figure 1 Input instance I of the burglar example

▶ **Example 2.** Our example is based on the burglar example of Pearl [39] that has been frequently used to illustrate probabilistic programming (e.g., [36]). Consider the EDB schema $\mathcal{E}$ consisting of the following relations: $\mathrm{House}(h, c)$ represents houses h and their location cities c, $\mathrm{Business}(b, c)$ represents businesses b and their location cities c, $\mathrm{City}(c, r)$ represents cities c and their associated burglary rates r, and $\mathrm{AlarmOn}(x)$ represents units (houses or businesses) x where the alarm is on. Figure 1 shows an instance I over this schema. Now consider the GDatalog[Δ] program $\mathcal{G} = (\mathcal{E}, \mathcal{I}, \Theta)$ of Figure 2.

Here, Δ consists of only one distribution, namely Flip. The first rule in Figure 2, intuitively, states that, for every fact of the form $\mathrm{City}(c, r)$, there must be a fact $\mathrm{Earthquake}(c, y)$ where y is drawn from the Flip (Bernoulli) distribution with the parameter 0.01. Moreover, the additional arguments Earthquake and c given after the semicolon (where Earthquake is a constant) enforce that different samples are drawn from the distribution for different cities (even if they have the same burglary rate), and that we never use the same sample as in Rules 5 and 6. Similarly, the presence of the additional argument x in Rule 4 enforces that a different sample is drawn for a different unit, instead of sampling only once per city.

▶ **Example 3.** The program of Figure 3 models virus dissemination among computers of email users. For simplicity, we identify each user with a distinct computer. Every message has a probability of passing a virus, if the virus is active on the source. If a message passes the virus, then the recipient has the virus (but it is not necessarily active, e.g., since the computer has the proper defence). And every user has a probability of having the virus active on her computer, in case she has the virus. Our program has the following EDBs:

- $\mathrm{Message}(m, s, t)$ contains message identifiers m sent from the user s to the user t.
- $\mathrm{VirusSource}(x)$ contains the users who are known to be virus sources.

In addition, the following IDBs are used.

- $\mathrm{PassVirus}(m, b)$ determines whether a message m passes a virus ($b = 1$) or not ($b = 0$).
- $\mathrm{HasVirus}(x, b)$ determines whether user x has the virus ($b = 1$) or not ($b = 0$).
- $\mathrm{ActiveVirus}(x, b)$ determines whether user x has the virus active ($b = 1$) or not ($b = 0$).

```
1. $\mathrm{Earthquake}(c, \mathsf{Flip}\llbracket 0.01; \mathsf{Earthquake}, c\rrbracket) \leftarrow \mathrm{City}(c, r)$
2. $\mathrm{Unit}(h, c) \leftarrow \mathrm{House}(h, c)$
3. $\mathrm{Unit}(b, c) \leftarrow \mathrm{Business}(b, c)$
4. $\mathrm{Burglary}(x, c, \mathsf{Flip}\llbracket r; \mathsf{Burglary}, x, c\rrbracket) \leftarrow \mathrm{Unit}(x, c)\,,\, \mathrm{City}(c, r)$
5. $\mathrm{Trig}(x, \mathsf{Flip}\llbracket 0.6; \mathsf{Trig}, x\rrbracket) \leftarrow \mathrm{Unit}(x, c)\,,\, \mathrm{Earthquake}(c, 1)$
6. $\mathrm{Trig}(x, \mathsf{Flip}\llbracket 0.9; \mathsf{Trig}, x\rrbracket) \leftarrow \mathrm{Burglary}(x, c, 1)$
7. $\mathrm{Alarm}(x) \leftarrow \mathrm{Trig}(x, 1)$
```

Figure 2 GDatalog[Δ] program $\mathcal{G}$ for the burglar example

1. $\mathrm{PassVirus}(m, \mathsf{Flip}[\![0.1; m]\!]) \leftarrow \mathrm{Message}(m, s, t),\ \mathrm{ActiveVirus}(s, 1)$
2. $\mathrm{HasVirus}(t) \leftarrow \mathrm{PassVirus}(m, 1),\ \mathrm{Message}(m, s, t)$
3. $\mathrm{ActiveVirus}(x, \mathsf{Flip}[\![0.5; x]\!]) \leftarrow \mathrm{HasVirus}(x)$
4. $\mathrm{ActiveVirus}(x, 1) \leftarrow \mathrm{VirusSource}(x)$

$$\begin{tikzcd}
\text{PassVirus,1} & \text{PassVirus,2} \\
\text{ActiveVirus,1} & \text{ActiveVirus,2} \\
 & \text{HasVirus,1} \arrow[ul] \arrow[u, "*"]
\end{tikzcd}$$

Figure 3 Program and dependency graph for the virus-dissemination example

The dependency graph depicted in Figure 3 will be used later on, in Section 3.4, when we further analyse this program.

Syntactic sugar. The syntax of GDatalog[Δ], as defined above, requires us to always make explicit the arguments that determine when different samples are taken from a distribution (cf. the argument c after the semicolon in Rule 1 of Figure 2, and the arguments x, c after the semicolon in Rule 4 of the same program). To enable a more succinct notation, we use the following convention: consider a Δ-atom $R(t_1, \dots, t_n)$ in which the i-th argument, t_i, is a Δ-term. Then t_i may be written using the simpler notation $\delta[\mathbf{p}]$, in which case it is understood to be a shorthand for $\delta[\![\mathbf{p}; \mathbf{q}]\!]$ where $\mathbf{q}$ is the sequence of terms $\mathsf{r}, i, t_1, \dots, t_{i-1}, t_{i+1}, \dots, t_n$. Here, r is a constant uniquely associated to the relation R. Thus, for example, $\mathrm{Earthquake}(c, \mathsf{Flip}[0.01]) \leftarrow \mathrm{City}(c, r)$ is taken to be a shorthand for $\mathrm{Earthquake}(c, \mathsf{Flip}[\![0.01; \mathsf{Earthquake}, 2, c]\!]) \leftarrow \mathrm{City}(c, r)$. Using this syntactic sugar, the program in Figure 2 can be rewritten in a notationally less verbose way, cf. Figure 4. Note, however, that the shorthand notation is less explicit as to describing when two rules involve the same sample vs. different samples from the same probability distribution.

3.2 Possible Outcomes

A GDatalog[Δ] program $\mathcal{G} = (\mathcal{E}, \mathcal{I}, \Theta)$ is associated with a corresponding Datalog$^{\exists}$ program $\widehat{\mathcal{G}} = (\mathcal{E}, \mathcal{I}^{\Delta}, \Theta^{\Delta})$. The *possible outcomes* of an input instance I w.r.t. $\mathcal{G}$ will then be minimal solutions of I w.r.t. $\widehat{\mathcal{G}}$. Next, we describe $\mathcal{I}^{\Delta}$ and Θ^{Δ}.

The schema $\mathcal{I}^{\Delta}$ extends $\mathcal{I}$ with the following additional relation symbols: for each $\delta \in \Delta$ with $\mathsf{pardim}(\delta) = k$ and for each $n \geq 0$, we have a $(k + n + 1)$-ary relation symbol $\mathrm{Result}^{\delta}_{n}$. These relation symbols $\mathrm{Result}^{\delta}_{n}$ are called the *distributional* relation symbols of $\mathcal{I}^{\Delta}$, and the other relation symbols of $\mathcal{I}^{\Delta}$ (namely, those of $\mathcal{I}$) are referred to as the *ordinary* relation symbols. Intuitively, a fact in $\mathrm{Result}^{\delta}_{n}$ represents the result of a particular sample drawn from δ (where k is the number of parameters of δ and n is the number of optional arguments that form the event signature).

The set Θ^{Δ} contains all Datalog rules from Θ that have no Δ-terms. In addition, for every

1. $\mathrm{Earthquake}(c, \mathsf{Flip}[0.01]) \leftarrow \mathrm{City}(c, r)$
2. $\mathrm{Unit}(h, c) \leftarrow \mathrm{House}(h, c)$
3. $\mathrm{Unit}(b, c) \leftarrow \mathrm{Business}(b, c)$
4. $\mathrm{Burglary}(x, c, \mathsf{Flip}[r]) \leftarrow \mathrm{Unit}(x, c)\,,\ \mathrm{City}(c, r)$
5. $\mathrm{Trig}(x, \mathsf{Flip}[0.6]) \leftarrow \mathrm{Unit}(x, c)\,,\ \mathrm{Earthquake}(c, 1)$
6. $\mathrm{Trig}(x, \mathsf{Flip}[0.9]) \leftarrow \mathrm{Burglary}(x, c, 1)$
7. $\mathrm{Alarm}(x) \leftarrow \mathrm{Trig}(x, 1)$

Figure 4 Burglar program from Figure 2 modified to use syntactic sugar

1a $\exists y\ \mathrm{Result}_2^{\mathsf{Flip}}(0.01, \mathsf{Earthquake}, c, y) \ \leftarrow\ \mathrm{City}(c, r)$
1b $\mathrm{Earthquake}(c, y) \ \leftarrow\ \mathrm{City}(c, r), \mathrm{Result}_2^{\mathsf{Flip}}(0.01, \mathsf{Earthquake}, c, y)$
2 $\mathrm{Unit}(h, c) \ \leftarrow\ \mathrm{House}(h, c)$
3 $\mathrm{Unit}(b, c) \ \leftarrow\ \mathrm{Business}(b, c)$
4a $\exists y\ \mathrm{Result}_3^{\mathsf{Flip}}(r, \mathsf{Burglary}, x, c, y) \ \leftarrow\ \mathrm{Unit}(x, c)\,,\ \mathrm{City}(c, r)$
4b $\mathrm{Burglary}(x, c, y) \ \leftarrow\ \mathrm{Unit}(x, c)\,,\ \mathrm{City}(c, r), \mathrm{Result}_3^{\mathsf{Flip}}(r, \mathsf{Burglary}, x, c, y)$
5a $\exists y \mathrm{Result}_2^{\mathsf{Flip}}(0.6, \mathsf{Trig}, x, y) \ \leftarrow\ \mathrm{Unit}(x, c)\,,\ \mathrm{Earthquake}(c, 1)$
5b $\mathrm{Trig}(x, y) \ \leftarrow\ \mathrm{Unit}(x, c)\,,\ \mathrm{Earthquake}(c, 1), \mathrm{Result}_2^{\mathsf{Flip}}(0.6, \mathsf{Trig}, y, x)$
6a $\exists y \mathrm{Result}_2^{\mathsf{Flip}}(0.9, \mathsf{Trig}, x, y) \ \leftarrow\ \mathrm{Burglary}(x, c, 1)$
6b $\mathrm{Trig}(x, y) \ \leftarrow\ \mathrm{Burglary}(x, c, 1), \mathrm{Result}_2^{\mathsf{Flip}}(0.9, \mathsf{Trig}, x, y)$
7 $\mathrm{Alarm}(x) \ \leftarrow\ \mathrm{Trig}(x, 1)$

Figure 5 The Datalog$^{\exists}$ program $\widehat{\mathcal{G}}$ for the GDatalog[Δ] program $\mathcal{G}$ of Figure 2

rule of the form $\psi(\mathbf{x}) \leftarrow \phi(\mathbf{x})$ in Θ, where ψ contains a Δ-term of the form $\delta[\![\mathbf{p};\mathbf{q}]\!]$ with $n = |\mathbf{q}|$, Θ^{Δ} contains the rules $\exists y \mathrm{Result}_n^{\delta}(\mathbf{p}, \mathbf{q}, y) \leftarrow \phi(\mathbf{x})$ and $\psi'(\mathbf{x}, y) \leftarrow \phi(\mathbf{x}), \mathrm{Result}_n^{\delta}(\mathbf{p}, \mathbf{q}, y)$, where ψ' is obtained from ψ by replacing $\delta[\![\mathbf{p};\mathbf{q}]\!]$ by y.

A *possible outcome* is defined as follows.

▶ **Definition 4** (Possible Outcome)**.** Let I be an input instance for a GDatalog[Δ] program $\mathcal{G}$. A *possible outcome* for I w.r.t. $\mathcal{G}$ is a minimal solution F of I w.r.t. $\widehat{\mathcal{G}}$, such that $\delta(b|\mathbf{p}) > 0$ for every distributional fact $\mathrm{Result}_n^{\delta}(\mathbf{p}, \mathbf{q}, b) \in F$.

We denote the set of all possible outcomes of I w.r.t. $\mathcal{G}$ by $\Omega_{\mathcal{G}}(I)$, and we denote the set of all finite possible outcomes by $\Omega_{\mathcal{G}}^{\mathsf{fin}}(I)$.

▶ **Example 5.** The GDatalog[Δ] program $\mathcal{G}$ given in Example 2 gives rise to the Datalog$^{\exists}$ program $\widehat{\mathcal{G}}$ of Figure 5. For instance, Rule 6 of Figure 2 is replaced with Rules 6a and 6b of Figure 5. An example of a possible outcome for the input instance I is the instance consisting of the relations in Figure 6 (ignoring the "pr(f)" columns for now), together with the relations of I itself.

3.3 Probabilistic Semantics

The semantics of a GDatalog[Δ] program is a function that maps every input instance I to a probability distribution over $\Omega_{\mathcal{G}}(I)$. We now make this precise. For a distributional fact f of the form $\mathrm{Result}_n^{\delta}(\mathbf{p}, \mathbf{q}, a)$, the *probability* of f, denoted $\mathrm{pr}(f)$, is defined to be $\delta(a|\mathbf{p})$. For an ordinary (non-distributional) fact f, we define $\mathrm{pr}(f) = 1$. For a finite set F of facts, we denote by $\mathbf{P}(F)$ the product of the probabilities of all the facts in F:[3]

$$\mathbf{P}(F) \overset{\mathsf{def}}{=} \prod_{f \in F} \mathrm{pr}(f)$$

▶ **Example 6.** (continued) Let J be the instance that consists of all of the relations in Figures 1 and 6. As we already remarked, J is a possible outcome of I w.r.t. $\mathcal{G}$. For convenience, in the case of distributional relations, we have indicated the probability of each fact next to the corresponding row. $\mathbf{P}(J)$ is the product of all of the numbers in the columns titled "pr(f)," that is, $0.01 \times 0.99 \times 0.9 \times \cdots \times 0.99$.

[3] The product reflects the law of total probability and does *not* assume that different random choices are independent (and indeed, correlation is clear in the examples throughout the paper).

$\text{Result}_2^{\mathsf{Flip}}$

p	att_1	att_2	$result$	$\text{pr}(f)$
0.01	Earthquake	Napa	1	0.01
0.01	Earthquake	Yucaipa	0	0.99
0.9	Trig	NP1	1	0.9
0.9	Trig	NP3	0	0.1
0.6	Trig	NP1	1	0.6
0.6	Trig	NP2	1	0.6
0.6	Trig	NP3	0	0.4

Unit

id	$city$
NP1	Napa
NP2	Napa
NP3	Napa
YC1	Yucaipa

Earthquake

$city$	eq
Napa	1
Yucaipa	0

Alarm

$unit$
NP1
NP2

$\text{Result}_3^{\mathsf{Flip}}$

p	att_1	att_2	att_3	$result$	$\text{pr}(f)$
0.03	Burglary	NP1	Napa	1	0.03
0.03	Burglary	NP2	Napa	0	0.97
0.03	Burglary	NP3	Napa	1	0.03
0.01	Burglary	YC1	Yucaipa	0	0.99

Burglary

$unit$	$city$	$draw$
NP1	Napa	1
NP2	Napa	0
NP3	Napa	1
YC1	Yucaipa	0

Trig

$unit$	$Trig$
NP1	1
NP3	0
NP2	1
NP3	0

Figure 6 A possible outcome for the input instance I in the burglar example

One can easily come up with examples where possible outcomes are infinite, and in fact, the space $\Omega_{\mathcal{G}}(I)$ of all possible outcomes is uncountable. Hence, we need to consider probability spaces over uncountable domains; those are defined by means of measure spaces.

Let $\mathcal{G}$ be a GDatalog[Δ] program, and let I be an input for $\mathcal{G}$. We say that a finite sequence $\mathbf{f} = (f_1, \dots, f_n)$ of facts is a *derivation* (w.r.t. I) if for all $i = 1, \dots, n$, the fact f_i is the result of applying some rule of $\mathcal{G}$ that is not satisfied in $I \cup \{f_1, \dots, f_{i-1}\}$ (in the case of applying a rule with a Δ-atom in the head, choosing a value randomly). If $f_1, \dots, f_n$ is a derivation, then the set $\{f_1, \dots, f_n\}$ is a *derivation set*. Hence, a finite set F of facts is a derivation set if and only if $I \cup F$ is an intermediate instance in some chase tree.

Let $\mathcal{G}$ be a GDatalog[Δ] program, I be an input for $\mathcal{G}$, and F be a set of facts. We denote by $\Omega_{\mathcal{G}}^{F\subseteq}(I)$ the set of all possible outcomes $J \subseteq \Omega_{\mathcal{G}}(I)$ such that $F \subseteq J$. The following theorem states how we determine the probability space defined by a GDatalog[Δ] program.

▶ **Theorem 7.** *Let $\mathcal{G}$ be a GDatalog[Δ] program, and let I be an input for $\mathcal{G}$. There exists a* unique *probability measure space $(\Omega, \mathcal{F}, \pi)$, denoted $\mu_{\mathcal{G},I}$, that satisfies all of the following.*

1. *$\Omega = \Omega_{\mathcal{G}}(I)$;*
2. *$(\Omega, \mathcal{F})$ is the σ-algebra generated from the sets of the form $\Omega_{\mathcal{G}}^{F\subseteq}(I)$ where F is finite;*
3. *$\pi(\Omega_{\mathcal{G}}^{F\subseteq}(I)) = \mathbf{P}(F)$ for every derivation set F.*

Moreover, if J is a finite possible outcome, then $\pi(\{J\})$ is equal to $\mathbf{P}(J)$.

Theorem 7 provides us with a semantics for GDatalog[Δ] programs: the semantics of a GDatalog[Δ] program $\mathcal{G}$ is a map from input instances I to probability measure spaces $\mu_{\mathcal{G},I}$ over possible outcomes (as uniquely determined by Theorem 7). The proof of Theorem 7 is by means of the *chase procedure*, which we discuss in the next section. A direct corollary of the theorem applies to the important case where all possible outcomes are finite (and the probability space may be infinite, but necessarily discrete).

▶ **Corollary 8.** *Let $\mathcal{G}$ be a GDatalog[Δ] program, and I an input instance for $\mathcal{G}$, such that $\Omega_{\mathcal{G}}(I) = \Omega_{\mathcal{G}}^{\mathsf{fin}}(I)$. Then $\mathbf{P}$ is a discrete probability function over $\Omega_{\mathcal{G}}(I)$; that is, $\sum_{J \in \Omega_{\mathcal{G}}(I)} \mathbf{P}(J) = 1$.*

3.4 Finiteness and Weak Acyclicity

Corollary 8 applies only when all solutions are finite, that is, $\Omega_{\mathcal{G}}(I) = \Omega^{\mathsf{fin}}_{\mathcal{G}}(I)$. We now present the notion of *weak acyclicity* for a GDatalog[Δ] program, as a natural syntactic condition that guarantees finiteness of all possible outcomes (for all input instances). This draws on the notion of weak acyclicity for Datalog$^{\exists}$ [18]. Consider any GDatalog[Δ] program $\mathcal{G} = (\mathcal{E}, \mathcal{I}, \Theta)$. A *position* of $\mathcal{I}$ is a pair (R, i) where $R \in \mathcal{I}$ and i is an attribute of R. The *dependency graph* of $\mathcal{G}$ is the directed graph that has the positions of $\mathcal{I}$ as the nodes, and the following edges:

- A *normal edge* $(R, i) \to (S, j)$ whenever there is a rule $\psi(\mathbf{x}) \leftarrow \varphi(\mathbf{x})$ and a variable x occurring at position (R, i) in $\varphi(\mathbf{x})$, and at position (S, j) in $\psi(\mathbf{x})$.
- A *special edge* $(R, i) \to^{*} (S, j)$ whenever there is a rule of the form

$$S(t_1, \ldots, t_{j-1}, \delta[\![\mathbf{p}; \mathbf{q}]\!], t_{j+1}, \ldots, t_n) \leftarrow \varphi(\mathbf{x})$$

  and a variable x occurring at position (R, i) in $\varphi(\mathbf{x})$ as well as in $\mathbf{p}$ or $\mathbf{q}$.

We say that $\mathcal{G}$ is *weakly acyclic* if no cycle in its dependency graph contains a special edge.

▶ **Theorem 9.** *If a GDatalog*[Δ] *program $\mathcal{G}$ is weakly acyclic, then $\Omega_{\mathcal{G}}(I) = \Omega^{\mathsf{fin}}_{\mathcal{G}}(I)$ for all input instances I.*

▶ **Example 10.** The burglar example program in Figure 2 is easily seen to be weakly acyclic (indeed, every non-recursive GDatalog[Δ] program is weakly-acyclic). In the case of the virus-dissemination example, the dependency graph in Figure 3 shows that, although this program features recursion, it is weakly acyclic as well.

3.5 Discussion

We conclude this section with some comments. First, we note that the restriction of a conclusion of a rule to include a single Δ-term significantly simplifies the presentation, but does not reduce the expressive power. In particular, we could simulate multiple Δ-terms in the conclusion using a collection of predicates and rules. For example, if one wishes to have conclusion where a person gets both a random height and a random weight (possibly with shared parameters), then she can do so by deriving $\mathrm{PersonHeight}(p, h)$ and $\mathrm{PersonWeight}(p, w)$ separately, and using the rule $\mathrm{PersonHW}(p, h, w) \leftarrow \mathrm{PersonHeight}(p, h), \mathrm{PersonWeight}(p, w)$. We also highlight the fact that our framework can easily simulate the probabilistic database model of *independent tuples* [45] with probabilities mentioned in the database. The framework can also simulate Bayesian networks, given relations that store the conditional probability tables, using the appropriate numerical distributions (e.g., Flip for the case of Boolean random variables). In addition, we note that a *disjunctive* Datalog rule [16], where the conclusion can be a disjunction of atoms, can be simulated by our model (with probabilities ignored): If the conclusion has n disjuncts, then we construct a distributional rule with a probability distribution over $\{1, \ldots, n\}$, and additional n deterministic rules corresponding to the atoms.

4 Chasing Generative Programs

The chase [3, 29] is a classic technique used for reasoning about database integrity constraints such as *tuple-generating dependencies*. This technique can be equivalently viewed as a tableaux-style proof system for $\forall^{*}\exists^{*}$-Horn sentences. In the special case of *full* tuple-generating dependencies, which are syntactically isomorphic to Datalog rules, the chase is closely related to (a tuple-at-a-time version of) the naive *bottom-up evaluation* strategy for Datalog program (cf. [2]). We now present a suitable variant of the chase for generative Datalog programs, and use it in order to construct the probability space of Theorem 7.

We note that, although the notions and results could arguably be phrased in terms of a probabilistic extension of the bottom-up Datalog evaluation strategy, the fact that a GDatalog[Δ] rule can create new values makes it more convenient to phrase them in terms of a suitable adaptation of the chase procedure.

Throughout this section, we fix a GDatalog[Δ] program $\mathcal{G} = (\mathcal{E}, \mathcal{I}, \Theta)$ and its associated Datalog$^{\exists}$ program $\widehat{\mathcal{G}} = (\mathcal{E}, \mathcal{I}^{\Delta}, \Theta^{\Delta})$. We first define the notions of *chase step* and *chase tree.*

Chase step. Consider an instance J, a rule $\tau \in \Theta^{\Delta}$ of the form $\psi(\mathbf{x}) \leftarrow \varphi(\mathbf{x})$, and a tuple $\mathbf{a}$ such that $\varphi(\mathbf{a})$ is satisfied in J but $\psi(\mathbf{a})$ is not satisfied in J. If $\psi(\mathbf{x})$ is a distributional atom of the form $\exists y \mathrm{Result}_i^{\delta}(\mathbf{p}, \mathbf{q}, y)$, then ψ being "not satisfied" is interpreted in the logical sense (regardless of probabilities): there is no y such that $(\mathbf{p}, \mathbf{q}, y)$ is in $\mathrm{Result}_i^{\delta}$. In that case, let $\mathcal{J}$ be the set of all instances J_b obtained by extending J with $\psi(\mathbf{a})$ for a specific value b of the existential variable y, such that $\delta(b|\mathbf{p}) > 0$. Furthermore, let π be the discrete probability distribution over $\mathcal{J}$ that assigns to J_b the probability $\delta(b|\mathbf{p})$. If $\psi(\mathbf{x})$ is an ordinary atom without existential quantifiers, $\mathcal{J}$ is simply defined as $\{J'\}$, where J' extends J with the fact $\psi(\mathbf{a})$, and $\pi(J') = 1$. We say that $J \xrightarrow{\tau(\mathbf{a})} (\mathcal{J}, \pi)$ is a *valid chase step.*

Chase tree. Let I be an input instance for $\mathcal{G}$. A *chase tree for I* w.r.t. $\mathcal{G}$ is a possibly infinite tree, whose nodes are labeled by instances over $\mathcal{E} \cup \mathcal{I}$, and whose edges are labeled by real numbers, such that:

1. The root is labeled by I;
2. For each non-leaf node labeled J, if $\mathcal{J}$ is the set of labels of the children of the node, and if π is the map assigning to each $J' \in \mathcal{J}$ the label of the edge from J to J', then $J \xrightarrow{\tau(\mathbf{a})} (\mathcal{J}, \pi)$ is a valid chase step for some rule $\tau \in \Theta^{\Delta}$ and tuple $\mathbf{a}$.
3. For each leaf node labeled J, there does not exist a valid chase step of the form $J \xrightarrow{\tau(\mathbf{a})} (\mathcal{J}, \pi)$. In other words, the tree cannot be extended to a larger chase tree.

We denote by $L(v)$ the label (instance) of the node v. Each $L(v)$ is said to be an *intermediate instance* w.r.t. the chase tree. Consider a GDatalog[Δ] program $\mathcal{G}$ and an input I for $\mathcal{G}$. A *maximal path* of a chase tree T is a path P that starts with the root, and either ends in a leaf or is infinite. Observe that the labels (instances) along a maximal path form a chain (w.r.t. the set-containment partial order). A maximal path P of a chase tree is *fair* if whenever the premise of a rule is satisfied by some tuple in some intermediate instance on P, then the conclusion of the rule is satisfied for the same tuple in some intermediate instance on P. A chase tree T is *fair* (or has the *fairness* property) if every maximal path is fair. Note that finite chase trees are fair. We restrict attention to fair chase trees. Fairness is a classic notion in the study of infinite computations;[4] moreover, fair chase trees can be constructed, for example, by maintaining a queue of "active rule firings."

Let $\mathcal{G}$ be a GDatalog[Δ] program, I be an input for $\mathcal{G}$, and T be a chase tree. We denote by *paths*(T) the set of maximal paths of T. (Note that *paths*(T) may be uncountably infinite.) For $P \in$ *paths*(T), we denote by $\cup P$ the union of the (chain of) labels $L(v)$ along P.

▶ **Theorem 11.** *Let $\mathcal{G}$ be a GDatalog[Δ] program, I an input for $\mathcal{G}$, and T a fair chase tree. The mapping $P \to \cup P$ is a bijection between paths(T) and $\Omega_{\mathcal{G}}(I)$.*

[4] Cf. any textbook on term rewriting systems or lambda calculus.

Chase measure. Let $\mathcal{G}$ be a GDatalog[Δ] program, let I be an input for $\mathcal{G}$, and let T be a chase tree. Our goal is to define a probability measure over $\Omega_{\mathcal{G}}(I)$. Given Theorem 11, we can do that by defining a probability measure over *paths*(T). A random path in *paths*(T) can be viewed as a *Markov chain* that is defined by a random walk over T, starting from the root. A measure space for such a Markov chain is defined by means of *cylindrification* [7]. Let v be a node of T. The v*-cylinder* of T, denoted C_v^T, is the subset of *paths*(T) that consists of all the maximal paths that contain v. A *cylinder* of T is a subset of *paths*(T) that forms a v-cylinder for some node v. We denote by $C(T)$ the set of all the cylinders of T.

Recall that $L(v)$ is a finite set of facts, and observe that $\mathbf{P}(L(v))$ is the product of the probabilities along the path from the root to v. The following theorem is a special case of a classic result on Markov chains (cf. [7]).

▶ **Theorem 12.** *Let $\mathcal{G}$ be a GDatalog[Δ] program, let I be an input for $\mathcal{G}$, and let T be a chase tree. There exists a unique probability measure $(\Omega, \mathcal{F}, \pi)$ that satisfies all of the following.*

1. *$\Omega =$ paths(T).*
2. *$(\Omega, \mathcal{F})$ is the σ-algebra generated from $C(T)$.*
3. *$\pi(C_v^T) = \mathbf{P}(L(v))$ for all nodes v of T.*

Theorems 11 and 12 suggest the following definition.

▶ **Definition 13** (Chase Probability Measure)**.** Let $\mathcal{G}$ be a GDatalog[Δ] program, let I be an input for $\mathcal{G}$, let T be a chase tree, and let $(\Omega, \mathcal{F}, \pi)$ be the probability measure of Theorem 12. The probability measure μ_T over $\Omega_{\mathcal{G}}(I)$ is the one obtained from $(\Omega, \mathcal{F}, \pi)$ by replacing every maximal path P with the possible outcome $\cup P$.

The following theorem states that the probability measure space represented by a chase tree is independent of the specific chase tree of choice.

▶ **Theorem 14.** *Let $\mathcal{G}$ be a GDatalog[Δ] program, let I be an input for $\mathcal{G}$, and let T and T' be two fair chase trees. Then $\mu_T = \mu_{T'}$.*

5 Probabilistic-Programming Datalog

To complete our framework, we define *probabilistic-programming Datalog*, *PPDL* for short, wherein a program augments a generative Datalog program with constraints; these constraints unify the traditional *integrity constraints* of databases and the traditional *observations* of probabilistic programming.

▶ **Definition 15** (PPDL[Δ])**.** Let Δ be a finite set of parameterized numerical distributions. A *PPDL*[Δ] *program* is a quadruple $(\mathcal{E}, \mathcal{I}, \Theta, \Phi)$, where $(\mathcal{E}, \mathcal{I}, \Theta)$ is a GDatalog[Δ] program and Φ is a finite set of logical constraints over $\mathcal{E} \cup \mathcal{I}$.[5]

▶ **Example 16.** Consider again Example 2. Suppose that we have the EDB relations ObservedHAlarm and ObservedBAlarm that represent observed home and business alarms, respectively. We obtain from the program in the example a PPDL[Δ]-program by adding the following constraints.

1. $\text{ObservedHAlarm}(h) \rightarrow \text{Alarm}(h)$

[5] We will address the choice of constraint language, and its algorithmic impact, in future work.

2. $\text{ObservedBAlarm}(b) \rightarrow \text{Alarm}(b)$
We use right (in contrast to left) arrows to distinguish constraints from ordinary Datalog rules. Note that a possible outcome J of an input instance I satisfies these constraints if J contains $\text{Alarm}(x)$ for all $x \in \text{ObservedHAlarm}^I \cup \text{ObservedBAlarm}^I$.

A PPDL[Δ] program defines the posterior distribution over its GDatalog[Δ] program, conditioned on the satisfaction of the constraints. A formal definition follows.

Let $\mathcal{P} = (\mathcal{E}, \mathcal{I}, \Theta, \Phi)$ be a PPDL[Δ] program, and let $\mathcal{G}$ be the GDatalog[Δ] program $(\mathcal{E}, \mathcal{I}, \Theta)$. An *input instance* for $\mathcal{P}$ is an input instance I for $\mathcal{G}$. We say that I is a *legal* input instance if $\{J \in \Omega_{\mathcal{G}}(I) \mid J \models \Phi\}$ is a measurable set in the probability space $\mu_{\mathcal{G},I}$, and its measure is nonzero. Intuitively, I is legal if it is consistent with the observations (i.e., with the constraints in Φ), given $\mathcal{G}$. The semantics of a PPDL[Δ] program is defined as follows.

▶ **Definition 17.** Let $\mathcal{P} = (\mathcal{E}, \mathcal{I}, \Theta, \Phi)$ be a PPDL[Δ] program, $\mathcal{G}$ the GDatalog[Δ] program $(\mathcal{E}, \mathcal{I}, \Theta)$, I a legal input instance for $\mathcal{P}$, and $\mu_{\mathcal{G},I} = (\Omega_{\mathcal{G}}(I), \mathcal{F}_{\mathcal{G}}, \pi_{\mathcal{G}})$. The probability space defined by $\mathcal{P}$ and I, denoted $\mu_{\mathcal{P},I}$, is the triple $(\Omega_{\mathcal{P}}(I), \mathcal{F}_{\mathcal{P}}, \pi_{\mathcal{P}})$ where:

1. $\Omega_{\mathcal{P}}(I) = \{J \in \Omega_{\mathcal{G}}(I) \mid J \models \Phi\}$
2. $\mathcal{F}_{\mathcal{P}} = \{S \cap \Omega_{\mathcal{P}}(I) \mid S \in \mathcal{F}_{\mathcal{G}}\}$
3. $\pi_{\mathcal{P}}(S) = \pi_{\mathcal{G}}(S)/\pi_{\mathcal{G}}(\Omega_{\mathcal{P}}(I))$ for every $S \in \mathcal{F}_{\mathcal{P}}$.
In other words, $\mu_{\mathcal{P},I}$ is $\mu_{\mathcal{G},I}$ conditioned on Φ.

▶ **Example 18.** Continuing Example 16, the semantics of this program is the posterior probability distribution that is obtained from the prior of Example 2, by conditioning on the fact that $\text{Alarm}(x)$ holds for all $x \in \text{ObservedHAlarm}^I \cup \text{ObservedBAlarm}^I$. Similarly, using an additional constraint we can express the condition that an alarm is off unless observed. One can ask various natural queries over this probability space of possible outcomes, such as the probability of the fact $\text{Earthquake}(\mathsf{Napa}, 1)$.

We note that when G is weakly acyclic, the event defined by Φ is measurable (since in that case the probability space is discrete) and the definition of legality boils down to the existence of a possible outcome.

5.1 Invariance under First-Order Equivalence

PPDL[Δ] programs are fully declarative in a strong sense: syntactically their rules and constraints can be viewed as first-order theories. Moreover, whenever two PPDL[Δ] programs, viewed in this way, are logically equivalent, then they are equivalent as PPDL[Δ] programs, in the sense that they give rise to the same set of possible outcomes and the same probability distribution over possible outcomes.

Formally, we say that two PPDL[Δ] programs, $\mathcal{P}_1 = (\mathcal{E}, \mathcal{I}, \Theta_1, \Phi_1)$ and $\mathcal{P}_2 = (\mathcal{E}, \mathcal{I}, \Theta_2, \Phi_2)$, are *semantically equivalent* if, for all input instances I, the probability spaces $\mu_{\mathcal{P}_1,I}$ and $\mu_{\mathcal{P}_2,I}$ coincide. Syntactically, the rules and constraints of a PPDL[Δ] program can be viewed as a finite first-order theory over a signature consisting of relation symbols, constant symbols, and function symbols (here, if the same name of a function name is used with different numbers of arguments, such as Flip in Figure 2, we treat them as distinct function symbols). We say that $\mathcal{P}_1$ and $\mathcal{P}_2$ are *FO-equivalent* if, viewed as first-order theories, Θ_1 is logically equivalent to Θ_2 (i.e., the two theories have the same models) and likewise for Φ_1 and Φ_2. We have the following theorems.

▶ **Theorem 19.** *If two PPDL[Δ] programs are FO-equivalent, then they are semantically equivalent (but not necessarily vice versa).*

▶ **Theorem 20.** *First-order equivalence is decidable for weakly acyclic GDatalog*$[\Delta]$ *programs. Semantic equivalence is undecidable for weakly acyclic GDatalog*$[\Delta]$ *programs (in fact, even for* $\Delta = \emptyset$*).*

6 Related Work

Our contribution is a marriage between probabilistic programming and the declarative specification of Datalog. The key features of our approach are the ability to express probabilistic models *concisely* and *declaratively* in a Datalog extension with probability distributions as first-class citizens. Existing formalisms that associate a probabilistic interpretation with logic are either not declarative (at least in the Datalog sense) or depart from the probabilistic programming paradigm (e.g., by lacking the support for numerical probability distributions). We next discuss representative related formalisms and contrast them with our work. They can be classified into three broad categories: *(1)* imperative specifications over logical structures, *(2)* logic over probabilistic databases, *and (3)* indirect specifications over the Herbrand base. (Some of these formalisms belong to more than one category.)

The first category includes imperative probabilistic programming languages [47]. We also include in this category declarative specifications of Bayesian networks, such as BLOG [32] and P-log [8]. Although declarative in nature, these languages inherently assume a form of acyclicity that allows the rules to be executed serially. Here we are able to avoid such an assumption since our approach is based on the minimal solutions of an existential Datalog program. The program in Figure 3, for example, uses recursion (as is typically the case for probabilistic models in social network analysis). In particular, it is not clear how this program can be phrased by translation into a Bayesian network. BLOG can express probability distributions over logical structures, via generative stochastic models that can draw values at random from numerical distributions, and condition values of program variables on observations. In contrast with closed-universe languages such as SQL and logic programs, BLOG considers open-universe probability models that allow for uncertainty about the existence and identity of objects.

The formalisms in the second category view the generative part of the specification of a statistical model as a two-step process. In the first step, facts are randomly generated by a mechanism external to the program. In the second step, a logic program, such as Prolog [26] or Datalog [1], is evaluated over the resulting random structure. This approach has been taken by PRISM [43], the *Independent Choice Logic* [41], and to a large extent by *probabilistic databases* [45] and their semistructured counterparts [25]. We focus on a formalism that completely defines the statistical model, without referring to external processes. As an important example, in PPDL one can sample from distributions that have parameters that by themselves are randomly generated using the program. This is the common practice in Bayesian machine learning (e.g., logistic regression), but it is not clear how it can be done within approaches of the second category.

One step beyond the second category and closer to our work is taken by uncertainty-aware query languages for probabilistic data such as TriQL [48], I-SQL, and world-set algebra [4, 5]. The latter two are natural analogs to SQL and relational algebra for the case of incomplete information and probabilistic data [4]. They feature constructs such as `repair-key`, `choice-of`, `possible`, and `group-worlds-by` that can construct possible worlds representing all repairs of a relation w.r.t. key constraints, close the possible worlds by unioning or intersecting them, or group the worlds into sets with the same results to sub-queries. World-set algebra has been extended to (world-set) Datalog, fixpoint, and while-languages [14] to define Markov chains. While such languages cannot explicitly specify probability distributions, they may simulate a specific categorical distribution indirectly using non-trivial programs with specialized language constructs like `repair-key` on input tuples with weights representing samples from the distribution.

MCDB [24] and SimSQL [12] propose SQL extensions (with for-loops and probability distributions) coupled with Monte Carlo simulations and parallel database techniques for stochastic analytics in the database. Their formalism does not involve the semantic challenges that we have faced in this paper. Although being based on SQL, these extensions do not offer a truly declarative means to specify probabilistic models, and end up being more similar to the imperative languages mentioned under the first category.

Formalisms in the third category use rule weighting as indirect specifications of probability spaces over the *Herbrand base*, which is the set of all the facts that can be obtained using the predicate symbols and the constants of the database. This category includes *Markov Logic Networks (MLNs)* [15,34], where the logical rules are used as a compact and intuitive way of defining *factors*. In other words, the probability of a possible world is the product of all the numbers (factors) that are associated with the grounded rules that the world satisfies. This approach is applied in DeepDive [35], where a database is used for storing relational data and extracted text, and database queries are used for defining the factors of a factor graph. We view this approach as *indirect* since a rule does not determine directly the distribution of values. Moreover, the semantics of rules is such that the addition of a rule that is logically equivalent to (or implied by, or indeed equal to) an existing rule changes the semantics and thus the probability distribution. A similar approach is taken by *Probabilistic Soft Logic* [11], where in each possible world every fact is associated with a degree of truth.

Further formalisms in this category are *probabilistic Datalog* [19], *probabilistic Datalog+/-* [21], and *probabilistic logic programming (ProbLog)* [26]. There, every rule is associated with a probability. For ProbLog, the semantics is not declarative as the rules follow a certain evaluation order; for probabilistic Datalog, the semantics is purely declarative. Both semantics are different from ours and that of the other formalisms mentioned thus far. A Datalog rule is interpreted as a rule over a probability distribution over possible worlds, and it states that, for a given grounding of the rule, the marginal probability of being true is as stated in the rule. Probabilistic Datalog+/- uses MLNs as the underlying semantics. Besides our support for numerical probability distributions, our formalism is used for defining a single probability space, which is in par with the standard practice in probabilistic programming.

As discussed earlier, GDatalog[Δ] allows for recursion, and the semantics is captured by (possibly infinite) Markov chains. Related formalisms are that of *Probabilistic Context-Free Grammars* (PCFG) and the more general *Recursive Markov Chains* (RMC) [17], where the probabilistic specification is by means of a finite set of transition graphs that can call one another (in the sense of method calls) in a possibly recursive fashion. In the database literature, PCFGs and RMCs are used in the context of probabilistic XML [9,13]. These formalisms do not involve numerical distributions. In future work, we plan to study their relative expressive power compared to restrictions of our framework.

7 Concluding Remarks

We proposed and investigated a declarative framework for specifying statistical models in the context of a database, based on a conservative extension of Datalog with numerical distributions. The framework differs from existing probabilistic programming languages not only due to the tight integration with a database, but also because of its fully declarative rule-based language: the interpretation of a program is independent of transformations (such as reordering or duplication of rules) that preserve the first-order semantics. This was achieved by treating a GDatalog[Δ] program as a Datalog program with existentially quantified variables in the conclusion of rules, and applying a suitable variant of the chase.

This paper opens various important directions for future work. One direction is to establish tractable conditions that guarantee that a given input is legal. Also, an interesting problem is to detect conditions under which the chase is a *self conjugate* [42], that is, the probability space $\mu_{\mathcal{P},I}$ is captured by a chase procedure without backtracking.

Our ultimate goal is to develop a full-fledged PP system based on the declarative specification language that we proposed here. In this work we focused on the foundations and robustness of the specification language. As in other PP languages, inference, such as computing the marginal probability of an IDB fact, is a challenging aspect, and we plan to investigate the application of common approaches such as sampling-based and lifted-inference techniques. We believe that the declarative nature of PPDL can lead to identifying interesting fragments that admit tractable complexity due to specialized techniques, just as is the case for Datalog evaluation in databases.

Practical applications will require further extensions to the language. We plan to support continuous probability distributions (e.g., continuous uniform, Pareto, and Gaussian), which are often used in statistical models. Syntactically, this extension is straightforward: we just need to include these distributions in Δ. Likewise, extending the probabilistic chase is also straightforward. More challenging is the semantic analysis, and, in particular, the definition of the probability space induced by the chase. We also plan to extend PPDL to support distributions that take a variable (and unbounded) number of parameters. A simple example is the *categorical* distribution where a single member of a finite domain of items is to be selected, each item with its own probability; in this case we can adopt the `repair-key` operation of the world-set algebra [4, 5]. Finally, we plan to add support for *multivariate distributions*, which are distributions with a support in $\mathbb{R}^k$ for $k > 1$ (where, again, k can be variable and unbounded). Examples of popular such distributions are multinomial, Dirichlet, and multivariate Gaussian distribution.

At LogicBlox, we are working on extending LogiQL with PPDL. An interesting syntactic and semantic challenge is that a program should contain rules of two kinds: probabilistic programming (i.e., PPDL rules) and inference over probabilistic programs (e.g., find the most likely execution). The latter rules involve the major challenge of efficient inference over PPDL. Towards that, our efforts fall in three different directions. First, we implement samplers of random executions. Second, we translate programs of restricted fragments into external statistical solvers (e.g., Bayesian Network libraries and *sequential Monte Carlo* [44]). Third, we are looking into fragments where we can apply exact and efficient (lifted) inference.

Acknowledgments

We are thankful to Molham Aref, Todd J. Green and Emir Pasalic for insightful discussions and feedback on this work. We also thank Michael Benedikt, Georg Gottlob and Yannis Kassios for providing useful comments and suggestions. Benny Kimelfeld is a Taub Fellow, supported by the Taub Foundation. Kimelfeld's work was supported in part by the Israel Science Foundation. Dan Olteanu acknowledges the support of the EPSRC programme grant VADA. This work was supported by DARPA under agreements #FA8750-15-2-0009 and #FA8750-14-2-0217. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright thereon.

References

1 Serge Abiteboul, Daniel Deutch, and Victor Vianu. Deduction with contradictions in Datalog. In *ICDT*, pages 143–154, 2014.

2 Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

3 Alfred V. Aho, Catriel Beeri, and Jeffrey D. Ullman. The theory of joins in relational databases. *ACM Trans. on Datab. Syst.*, 4(3):297–314, 1979.

4 Lyublena Antova, Christoph Koch, and Dan Olteanu. From complete to incomplete information and back. In *SIGMOD*, pages 713–724, 2007.

5 Lyublena Antova, Christoph Koch, and Dan Olteanu. Query language support for incomplete information in the MayBMS system. In *VLDB*, pages 1422–1425, 2007.

6 Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. Design and implementation of the LogicBlox system. In *PDOS*, pages 1371–1382. ACM, 2015.

7 Robert B. Ash and Catherine Doleans-Dade. *Probability & Measure Theory*. Harcourt Academic Press, 2000.

8 Chitta Baral, Michael Gelfond, and Nelson Rushton. Probabilistic reasoning with answer sets. *Theory Pract. Log. Program.*, 9(1):57–144, 2009.

9 Michael Benedikt, Evgeny Kharlamov, Dan Olteanu, and Pierre Senellart. Probabilistic XML via Markov chains. *PVLDB*, 3(1):770–781, 2010.

10 David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *J. of Machine Learning Research*, 3:993–1022, 2003.

11 Matthias Bröcheler, Lilyana Mihalkova, and Lise Getoor. Probabilistic similarity logic. In *UAI*, pages 73–82, 2010.

12 Zhuhua Cai, Zografoula Vagena, Luis Leopoldo Perez, Subramanian Arumugam, Peter J. Haas, and Christopher M. Jermaine. Simulation of database-valued Markov chains using SimSQL. In *SIGMOD*, pages 637–648, 2013.

13 Sara Cohen and Benny Kimelfeld. Querying parse trees of stochastic context-free grammars. In *ICDT*, pages 62–75. ACM, 2010.

14 Daniel Deutch, Christoph Koch, and Tova Milo. On probabilistic fixpoint and Markov chain query languages. In *PODS*, pages 215–226, 2010.

15 Pedro Domingos and Daniel Lowd. *Markov Logic: An Interface Layer for Artificial Intelligence*. Synthesis Lectures on AI and Machine Learning. Morgan & Claypool Publishers, 2009.

16 Thomas Eiter, Georg Gottlob, and Heikki Mannila. Disjunctive Datalog. *ACM Trans. Database Syst.*, 22(3):364–418, 1997.

17 Kousha Etessami and Mihalis Yannakakis. Recursive Markov chains, stochastic grammars, and monotone systems of nonlinear equations. *J. ACM*, 56(1), 2009.

18 Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. Data exchange: Semantics and query answering. In *ICDT*, pages 207–224, 2003.

19 Norbert Fuhr. Probabilistic Datalog: Implementing logical information retrieval for advanced applications. *JASIS*, 51(2):95–110, 2000.

20 Noah D. Goodman. The principles and practice of probabilistic programming. In *POPL*, pages 399–402, 2013.

21 Georg Gottlob, Thomas Lukasiewicz, MariaVanina Martinez, and Gerardo Simari. Query answering under probabilistic uncertainty in Datalog+/- ontologies. *Annals of Math.& AI*, 69(1):37–72, 2013.

22 Terry Halpin and Spencer Rugaber. *LogiQL: A Query Language for Smart Databases*. CRC Press, 2014.

23 Shan Shan Huang, Todd Jeffrey Green, and Boon Thau Loo. Datalog and emerging applications: An interactive tutorial. In *SIGMOD*, pages 1213–1216, 2011.

24 Ravi Jampani, Fei Xu, Mingxi Wu, Luis Leopoldo Perez, Christopher M. Jermaine, and Peter J. Haas. MCDB: a Monte Carlo approach to managing uncertain data. In *SIGMOD*, pages 687–700, 2008.

25 Benny Kimelfeld and Pierre Senellart. Probabilistic XML: models and complexity. In *Advances in Probabilistic Databases for Uncertain Information Management*, volume 304 of *Studies in Fuzziness and Soft Computing*, pages 39–66. Springer, 2013.

26 Angelika Kimmig, Bart Demoen, Luc De Raedt, Vitor Santos Costa, and Ricardo Rocha. On the implementation of the probabilistic logic programming language ProbLog. *Theory and Practice of Logic Programming*, 11:235–262, 2011.

27 Lyric Labs. Chimple. `http://chimple.probprog.org/`.

28 Boon Thau Loo, Tyson Condie, Minos N. Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative networking. *Commun. ACM*, 52(11):87–95, 2009.

29 David Maier, Alberto O. Mendelzon, and Yehoshua Sagiv. Testing implications of data dependencies. *ACM Trans. on Datab. Syst.*, 4(4):455–469, 1979.

30 Vikash K. Mansinghka, Daniel Selsam, and Yura N. Perov. Venture: a higher-order probabilistic programming platform with programmable inference. *CoRR*, abs/1404.0099, 2014.

31 Andrew Kachites McCallum. Multi-label text classification with a mixture model trained by EM. In *Assoc. for the Advancement of Artificial Intelligence workshop on text learning*, 1999.

32 B. Milch and et al. BLOG: Probabilistic models with unknown objects. In *IJCAI*, pages 1352–1359, 2005.

33 Kamal Nigam, Andrew McCallum, Sebastian Thrun, and Tom M. Mitchell. Text classification from labeled and unlabeled documents using EM. *Machine Learning*, pages 103–134, 2000.

34 Feng Niu, Christopher Ré, AnHai Doan, and Jude W. Shavlik. Tuffy: Scaling up statistical inference in Markov Logic Networks using an RDBMS. *PVLDB*, 4(6):373–384, 2011.

35 Feng Niu, Ce Zhang, Christopher Re, and Jude W. Shavlik. DeepDive: Web-scale knowledge-base construction using statistical learning and inference. In *Int. Workshop on Searching and Integrating New Web Data Sources*, volume 884 of *CEUR Workshop Proceedings*, pages 25–28, 2012.

36 Aditya V. Nori, Chung-Kil Hur, Sriram K. Rajamani, and Selva Samuel. R2: an efficient MCMC sampler for probabilistic programs. In *AAAI*, pages 2476–2482, 2014.

37 Brooks Paige and Frank Wood. A compilation target for probabilistic programming languages. In *ICML*, volume 32, pages 1935–1943, 2014.

38 Anand Patil, David Huard, and Christopher J. Fonnesbeck. PyMC: Bayesian Stochastic Modelling in Python. *J. of Statistical Software*, 35(4):1–81, 2010.

39 Judea Pearl. *Probabilistic reasoning in intelligent systems - networks of plausible inference.* Morgan Kaufmann, 1989.

40 Avi Pfeffer. Figaro: An object-oriented probabilistic programming language. Technical report, Charles River Analytics, 2009.

41 David Poole. The independent choice logic and beyond. In *Probabilistic Inductive Logic Programming - Theory and Applications*, pages 222–243, 2008.

42 H. Raiffa and R. Schlaifer. *Applied Statistical Decision Theory.* Harvard University Press, Harvard, 1961.

43 Taisuke Sato and Yoshitaka Kameya. PRISM: A language for symbolic-statistical modeling. In *IJCAI*, pages 1330–1339, 1997.

44 Adrian Smith, Arnaud Doucet, Nando de Freitas, and Neil Gordon. *Sequential Monte Carlo methods in practice*. Springer Science & Business Media, 2013.

45 Dan Suciu, Dan Olteanu, Christopher Ré, and Christoph Koch. *Probabilistic Databases*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011.

46 Balder ten Cate, Benny Kimelfeld, and Dan Olteanu. PPDL: probabilistic programming with Datalog. In *AMW*, volume 1378 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2015.

47 `http://www.probabilistic-programming.org`. Repository on probabilistic programming languages, 2014.

48 Jennifer Widom. Trio: a system for data, uncertainty, and lineage. In Charu Aggarwal, editor, *Managing and Mining Uncertain Data*, chapter 5. Springer-Verlag, 2008.