Declarative Probabilistic Programming with Datalog

VINCE BÁRÁNY and BALDER TEN CATE, LogicBlox, Inc.
BENNY KIMELFELD, Technion – Israel Institute of Technology
DAN OLTEANU, University of Oxford
ZOGRAFOULA VAGENA, LogicBlox, Inc.

Probabilistic programming languages are used for developing statistical models. 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.

In this article, we establish a probabilistic-programming extension of Datalog that, on the one hand, allows for defining a rich family of statistical models, and on the other hand retains the fundamental properties of declarativity. Our proposed extension provides 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.

CCS Concepts: • **Theory of computation** → **Constraint and logic programming**; **Database query languages (principles)**; • **Mathematics of computing** → *Probabilistic representations*;

Additional Key Words and Phrases: Chase, Datalog, declarative, probability measure space, probabilistic programming

ACM Reference format:
Vince Bárány, Balder Ten Cate, Benny Kimelfeld, Dan Olteanu, and Zografoula Vagena. 2017. Declarative Probabilistic Programming with Datalog. *ACM Trans. Database Syst.* 42, 4, Article 22 (October 2017), 35 pages.
https://doi.org/10.1145/3132700

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

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 program grant VADA number M025268. This work was supported by DARPA under agreements #FA8750-15-2-0009 and #FA8750-14-2-0217. The US government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright thereon.

Authors' addresses: V. Bárány and B. T. Cate, Google, Inc., 1600 Amphitheatre Pkwy, Mountain View, CA 94043, USA; emails: {vbarany, balder.tencate}@gmail.com; B. Kimelfeld, Taub 703, Faculty of Computer Science, Technion – Israel Institute of Technology, Haifa 32000, Israel; email: bennyk@cs.technion.ac.il; D. Olteanu, Wolfson Building, Parks Road, OXFORD OX1 3QD, UK; email: dan.olteanu@cs.ox.ac.uk; Z. Vagena, Infor Inc., 1349 West Peachtree Street, Atlanta, GA 30309, USA; email: zografoula.vagena@logicblox.com.

ACM 0362-5915/2017/10-ART22 $15.00
https://doi.org/10.1145/3132700

uncertainty. A substantial effort has been made in developing such formalisms and corresponding system implementations (Paige and Wood 2014; Hershey et al. 2012; Pfeffer 2009; Mansinghka et al. 2014; Patil et al. 2010; Cowles 2013; Carpenter et al. 2017). An actively studied concept in that area is that of *Probabilistic Programming* (PP) (Goodman 2013), 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 outcomes of the program, also referred to as *possible worlds*. Examples of such tasks include finding the most likely possible world or estimating the probability of a property of the outcome. Recently, DARPA initiated the project *Probabilistic Programming for Advancing Machine Learning (PPAML)*, aimed at advancing PP systems, focusing on a specific collection of systems (Pfeffer 2009; Mansinghka et al. 2014; Milch et al. 2005), toward 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 one 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/nonspam). 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. The common Bayesian 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 (Nigam et al. 2000; McCallum 1999). 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 that are not known in advance. Latent Dirichlet Allocation (Blei et al. 2003) approaches this problem in a similar generative way as the aforementioned, 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 (Huang et al. 2011). In each of these applications, the motivation is that being declarative, Datalog has the potential to make specifications easier to write (sometimes with orders-of-magnitude fewer lines of code than imperative code, e.g., Loo et al. (2009)), comprehend, and maintain, the canonical example being reachability or cyclicity testing in graphs.

Our goal in this article is to establish a probabilistic extension of Datalog that, on the one hand, allows for defining a rich family of statistical models, and on the other hand retains the fundamental properties of declarativity, namely, independence of the execution order and invariance under equivalence-preserving rewriting. On 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 (Paige and Wood 2014), Java (Hershey et al. 2012), Scala (Pfeffer 2009), Scheme (Mansinghka et al. 2014), and Python (Patil et al. 2010). We discuss the relationship of this work to related literature in Section 7. In the context of the LogicBlox system (Aref et al. 2015), we are interested in extending the Datalog-based LogiQL (Halpin and Rugaber 2014) with PP to enable and facilitate the development of predictive analysis. 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).

When attempting to extend Datalog with probabilistic programming constructs, a fundamental challenge 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 most efficient, and to apply techniques for partial evaluation of rules in incremental view maintenance (Gupta et al. 1993; Aref et al. 2015; Motik et al. 2015). Another inherent 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 into one that is more efficient to execute, 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: LivesIn(person, city), WorksFor(person, employer), LocatedIn(company, city), and AvgVisits(city, avg). The following rule provides an appealing way to model the generation of a random number of visits for a person:

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

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:

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

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 Poisson[λ].

Furthermore, consider the following (standard) rewriting of the previous program:

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

As a conjunction of first-order sentences that views Poisson[λ] as a function term, the rewritten program is logically equivalent to the previous one (made up of Equations (1) and (2)); 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* (Domingos and Lowd 2009) or *Probabilistic Soft Logic* (Bröcheler et al. 2010), this 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 article, our focus is on discrete numerical distributions. As we discuss in Section 8, the framework we introduce admits a natural generalization to continuous distributions, such as Gaussian or Pareto, but adjusting our theoretical analysis to such distributions is nontrivial since our proof techniques essentially rely on the discreteness of distributions. 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 article has appeared in the 19th International Conference on Database Theory (ICDT'16) (Bárány et al. 2016). This article has the following additions compared to the short version. First, it includes all the proofs and intermediate results, which were omitted from the short version. Second, it includes a new section, Section 6, that describes the translation of Markov Logic Networks and stochastic context-free grammars into PPDL.

The rest of the article is organized as follows. After presenting preliminary definitions and concepts (Section 2), we describe the generative component of PPDL that we term GDatalog (Section 3). We then present a chase procedure for GDatalog and use it to prove some fundamental results (Section 4). We complement GDatalog with constraints (or observations) to establish the PPDL language (Section 5). We illustrate PPDL by showing a general translation from *Markov Logic Networks* (MLNs) into PPDL, and a translation from *stochastic context-free grammars* into PPDL (Section 6). Finally, we discuss related work (Section 7) and conclude (Section 8).

2 PRELIMINARIES

In this section, we present preliminary notation and definitions that we use throughout the article.

2.1 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 $\{1, \ldots, \mathsf{arity}(R)\}$. 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 $R(c_1, \ldots, c_n)$, where R is an n-ary relation in $\mathcal{S}$ and $c_1, \ldots, c_n \in \mathbb{R}$. 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 $(c_1, \ldots, c_n)$ such that $R(c_1, \ldots, c_n) \in I$.

2.2 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 $\forall \mathbf{x}[\,(\exists \mathbf{y}\psi(\mathbf{x},\mathbf{y})) \leftarrow \varphi(\mathbf{x})\,]$, 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 $R(t_1, \ldots, t_n)$, where R is an n-ary relation and $t_1, \ldots, t_n$ 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* (i.e., without existential quantifiers), then every input instance I has a unique minimal solution, which is finite, and therefore $\mathsf{min\text{-}sol}^{\mathsf{fin}}_{\mathcal{D}}(I) = \mathsf{min\text{-}sol}_{\mathcal{D}}(I)$ (Abiteboul et al. 1995).

2.3 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 (nonparameterized) 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 8, 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 $\mathcal{F}$. For example, a discrete probability space (Ω', π') is viewed as the probability measure space $(\Omega, \mathcal{F}, \pi)$, where $\Omega' = \Omega$, the σ-algebra $\mathcal{F}$ consists of all the subsets of Ω, and π' is defined by $\pi'(e) = \sum_{o \in e} \pi(e)$.

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*

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

Fig. 1. Input instance I of the burglar example.

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. We begin by describing the syntax of generative Datalog programs, and then establish their semantics. Specifically, we describe the notion of *possible outcomes* that constitute the samples in the sample space, which may be uncountable. We then construct a probability measure space over the sample space. Our main result (Theorem 3.8) states that this probability measure space is well defined. The proof of this result is deferred to Section 4. Finally, we discuss a syntactic restriction, *weak acyclicity*, which guarantees the finiteness of possible outcomes.

3.1 Syntax

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

Definition 3.1 (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 $\mathsf{pardim}(\delta) = \ell \leq k$, 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 $\delta[\![\mathbf{p}; \mathbf{q}]\!]$ (with $\mathbf{q}$ being the event signature). 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 $\mathcal{I}$.

Example 3.2. Our example is based on the burglar example of Pearl (1989) that has been frequently used to illustrate probabilistic programming (e.g., Nori et al. (2014)). Consider the EDB schema $\mathcal{E}$ consisting of the following relations: House(h, c) represents houses h and their location cities c, Business(b, c) represents businesses b and their location cities c, City(c, r) represents cities c and their associated burglary rates r, and 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. Rule (1) in Figure 2, intuitively, states that, for every fact of the form City(c, r), there must be a fact Earthquake(c, y) where y is drawn from

[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 3.2).

[2]The restriction to at most one Δ-term per atom is only for presentational purposes (cf. Section 3.5).

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

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

the Flip (Bernoulli) distribution with the parameter 0.01. Rules (2) and (3) define Unit as the union of House and Business. Rule (4) states that a burglary in a unit x is drawn by a Flip distribution with the parameter r, which is the burglary rate of the city of x. Rules (5) and (6) state that an alarm is randomly triggered in a house if there is either an earthquake or a burglary; in the first case the probability is 0.6 and in the second 0.9. Finally, Rule (7) states that the alarm is on in unit x if it is triggered therein.

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 use samples different from those in Rules (5) and (6), as those use the constant Trig rather than Earthquake. 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 3.3. The program of Figure 4 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 the virus is not necessarily active, for instance, since the computer has the proper defense. And every user has a probability of having the virus active on his or her computer, in case he or she has the virus. Our program has the following EDBs:

— Message(m, s, t) contains message identifiers m sent from the user s to the user t.
— VirusSource(x) contains the users who are known to be virus sources.

In addition, the following IDBs are used:

— PassVirus(m, b) determines whether message m passes a virus ($b = 1$) or not ($b = 0$).
— HasVirus(x, b) determines whether user x has the virus ($b = 1$) or not ($b = 0$).
— ActiveVirus(x, b) determines whether the virus is active for user x ($b = 1$) or not ($b = 0$).

3.1.1 Syntactic Sugar. The syntax of GDatalog[Δ], as defined previously, 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 $R(t_1, \ldots, t_n)$ in which the ith 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 $\mathsf{r}, i, t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n$. Here, r is a constant uniquely associated with the relation R. Thus, for example, Earthquake$(c, \mathsf{Flip}[0.01])$ $\leftarrow$ City(c, r) is taken to be a shorthand for Earthquake$(c, \mathsf{Flip}[\![0.01; \mathsf{Earthquake}, c]\!])$ $\leftarrow$ City(c, r). Using this syntactic sugar, the program in Figure 2 can be rewritten in a notationally less verbose way (cf. Figure 3). Note, however, that the shorthand notation is less explicit as to describing when two rules involve the same sample versus different samples from the same probability distribution.

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

Fig. 3. Burglar program from Figure 2 modified to use syntactic sugar.

$$\begin{array}{|ll|}\hline
(1) & \mathrm{PassVirus}(m, \mathsf{Flip}[\![0.1; m]\!]) \;\leftarrow\; \mathrm{Message}(m, s, t),\ \mathrm{ActiveVirus}(s, 1) \\
(2) & \mathrm{HasVirus}(t) \;\leftarrow\; \mathrm{PassVirus}(m, 1),\ \mathrm{Message}(m, s, t) \\
(3) & \mathrm{ActiveVirus}(x, \mathsf{Flip}[\![0.5; x]\!]) \;\leftarrow\; \mathrm{HasVirus}(x) \\
(4) & \mathrm{ActiveVirus}(x, 1) \;\leftarrow\; \mathrm{VirusSource}(x) \\ \hline
\end{array}$$

Fig. 4. Virus dissemination example.

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 Result^δ_n. These relation symbols Result^δ_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 Result^δ_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 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 $\exists y \mathrm{Result}^\delta_n(\mathbf{p}, \mathbf{q}, y) \leftarrow \phi(\mathbf{x})$ and $\psi'(\mathbf{x}, y) \leftarrow \phi(\mathbf{x}), \mathrm{Result}^\delta_n(\mathbf{p}, \mathbf{q}, y)$, where ψ' is obtained from ψ by replacing $\delta[\![\mathbf{p}; \mathbf{q}]\!]$ with y.

A *possible outcome* is defined as follows.

Definition 3.4 (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 $\mathrm{Result}^\delta_n(\mathbf{p}, \mathbf{q}, b) \in F$.

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 $\Omega^{\mathsf{fin}}_{\mathcal{G}}(I)$.

Example 3.5. The GDatalog[Δ] program $\mathcal{G}$ given in Example 3.2 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.

The following proposition provides insight into the possible outcomes of an instance and will reappear later on in our study of the chase. For any distributional relation $R^\delta_n \in \Theta^\Delta$, the *functional dependency associated with* R^δ_n is the functional dependency $R^\delta_n : (\{1, \ldots, \mathsf{arity}(R^\delta_n) - 1\}) \rightarrow \mathsf{arity}(R^\delta_n)$, expressing that the last attribute of R^δ_n is functionally determined by the other attributes.

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

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

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

p	*att*$_1$	*att*$_2$	*result*	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

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

p	*att*$_1$	*att*$_2$	*att*$_3$	*result*	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

Fig. 6. A possible outcome for the input instance *I* in the burglar example.

Proposition 3.6. *Let I be any input instance for a GDatalog[Δ] instance $\mathcal{G}$. Then every possible outcome in $\Omega_{\mathcal{G}}(I)$ satisfies all functional dependencies associated with distributional relations.*

Proof. If an instance J violates the functional dependency associated with a distributional relation R_n^{δ}, then one of the two facts involved in the violation can be removed, showing that J is, in fact, not a minimal solution w.r.t. $\widehat{\mathcal{G}}$. □

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 $\Pr(f)$, is defined to be $\delta(a|\mathbf{p})$. For an ordinary (nondistributional) fact f, we define $\Pr(f) = 1$. 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]

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

Example 3.7 (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, $0.01 \times 0.99 \times 0.9 \times \cdots \times 0.99$.

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 $\mathbf{f} = (f_1, \ldots, f_n)$ of facts is a *derivation* (w.r.t. I) if for all $i = 1, \ldots, n$, the fact f_i is the result of applying some rule of $\mathcal{G}$ that is not satisfied in $I \cup \{f_1, \ldots, f_{i-1}\}$; in the case of applying a rule with a Δ-atom in the head, f_i contains a randomly chosen value. If $f_1, \ldots, f_n$ is a derivation, then the set $\{f_1, \ldots, f_n\}$ 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. The proof will be given in Section 4.2.

Theorem 3.8. *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 3.8 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—this is the measure space uniquely determined according to the theorem.

A direct corollary to Theorem 3.8 applies to the important case where all possible outcomes are finite. Note that in this case, the probability space may still be infinite, but it is necessarily discrete.

Corollary 3.9. *Let $\mathcal{G}$ be a GDatalog[Δ] program and I be an input instance for $\mathcal{G}$, such that $\Omega_{\mathcal{G}}(I) = \Omega_{\mathcal{G}}^{\mathrm{fin}}(I)$. 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]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 article).

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

Fig. 7. The dependency graph for the virus dissemination example of Figure 4.

In essence, Corollary 3.9 states that the function $\mathbf{P}(\cdot)$ forms a probability distribution over the possible outcomes, even though it is defined in an algebraic manner. In Section 4, we will associate this probability distribution with a natural stochastic process. The following section introduces a syntactic condition, *weak acyclicity*, that guarantees the finiteness of solutions. Note, however, that we *do not* assume finiteness or acyclicity in the article, unless explicitly stated.

3.4 Finiteness and Weak Acyclicity

Corollary 3.9 applies only when all solutions are finite, that is, $\Omega_{\mathcal{G}}(I) = \Omega^{\text{fin}}_{\mathcal{G}}(I)$. 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$^{\exists}$ (Fagin et al. 2005). Intuitively, weak acyclicity is a synthetic condition that guarantees that the program holds the property of *distribution stratification* defined for the *Distributional Clauses* (*DC*) (Gutmann et al. 2011); we further discuss DC and distribution stratification in Section 7.

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

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

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. The following theorem states that, as promised, weak acyclicity guarantees the finiteness of the possible outcomes. The proof is similar to the proof of the corresponding result of Fagin et al. (2005, Theorem 3.9). Moreover, as in their work, weak acyclicity of a program can be tested efficiently via graph reachability.

Theorem 3.10. *If a GDatalog[Δ] program $\mathcal{G}$ is weakly acyclic, then $\Omega_{\mathcal{G}}(I) = \Omega^{\text{fin}}_{\mathcal{G}}(I)$ for all input instances I.*

Next, we give an example.

Example 3.11. The burglar example program in Figure 2 is easily seen to be weakly acyclic (indeed, every nonrecursive GDatalog[Δ] program is weakly acyclic). In the case of the virus dissemination example of Figure 4, the dependency graph in Figure 7 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 a conclusion where a person gets both a random height and a random weight (possibly with shared parameters), then he or she can do so by deriving $\text{PersonHeight}(p, h)$ and $\text{PersonWeight}(p, w)$ separately, and using the rule $\text{PersonHW}(p, h, w) \leftarrow \text{PersonHeight}(p, h), \text{PersonWeight}(p, w)$. We also highlight the fact that our framework can easily simulate the probabilistic database model of *independent tuples* (Suciu et al. 2011) 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 (Eiter et al. 1997), 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 (Maier et al. 1979; Aho et al. 1979) 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 the Datalog program (cf. Abiteboul et al. (1995)). 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 3.8. Specifically, we first define the concepts of a *chase step* and *chase tree*. We then define a probability measure space by means of a random walk over the chase tree. We establish two main results. First, while chase trees for the same program can be different (due to different chasing orders), they define the same probability measure space (Theorem 4.7). Second, this measure space is the one that realizes Theorem 3.8 (Section 4.2).

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*. These are similar to the *probabilistic Σ-process* in CP-logic (Vennekens et al. 2009), which we discuss in more detail in Section 7, except that our chase trees can have infinite paths.

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 $\exists y \text{Result}_i^\delta(\mathbf{p}, \mathbf{q}, y)$, 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 Result_i^δ. 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 nonleaf 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 (Lobo et al. 1992; Lloyd 1987; Dershowitz 2005; Francez 1986); moreover, fair chase trees can be constructed, for example, by maintaining a queue of "active rule firings."

4.1 Properties of the Chase

In this section, we show several properties of our chase procedure. These properties are needed for later proofs.

A chase tree is said to be *injective* if no intermediate instance is the label of more than one node; that is, for $v_1 \neq v_2$ we have $L(v_1) \neq L(v_2)$. As we will see shortly, due to the specific construction of Θ^Δ, every chase tree turns out to be injective.

Proposition 4.1. *Every chase tree w.r.t. $\mathcal{G}$ is injective.*

Proof. For the sake of a contradiction, assume that two nodes n_1 and n_2 in a chase tree are labeled by the same instance J. Let n_0 be the node that is the least common ancestor of n_1 and n_2 in the tree, and let n'_1 and n'_2 be the children of n_0 that are ancestors of n_1 and n_2, respectively. By construction, n'_1 and n'_2 are labeled with distinct instances $J_1 \neq J_2$, respectively. Consider the rule $\tau = \psi(\mathbf{x}) \leftarrow \varphi(\mathbf{x})$ and tuple $\mathbf{a}$ constituting the chase step applied at node n_0. Since n_0 has more than one child, $\psi(\mathbf{x})$ must be a distributional atom, say, $\exists y R_i^\delta(\mathbf{t}, y, \mathbf{t}', \mathbf{p})$. Then each J_k ($k = 1, 2$) contains an R_i^δ-fact. Moreover, the two R_i^δ-facts in question differ in the choice of value for the variable y and are otherwise identical. Due to the monotonic nature of the chase, both atoms must belong J, and hence, J violates the functional dependency of Proposition 3.6. Hence, we have reached a contradiction. □

Proposition 4.2. *Let I be any input instance, and consider any chase tree for I w.r.t. $\mathcal{G}$. Then every intermediate instance satisfies all functional dependencies associated with distributional relations.*

Proof. The proof is by induction on the distance from the root of the chase tree. The basis is the root of the tree, where no IDB facts exist, and the functional dependencies are satisfied in a vacuous manner. For the inductive step, suppose that in a chase step $J \xrightarrow{\tau(\mathbf{a})} (\mathcal{J}, \pi)$, some $J' \in \mathcal{J}$ contains two R_i^δ-facts that are identical except for the ith attribute (recall that R_i^δ is a distributional atom of arity i). Then, we have two cases. Either J already contains both atoms, in which case we can apply our induction hypothesis, or J' is obtained by extending J with one of the two facts in question. In the latter case, the conclusion of τ was already satisfied for the tuple $\mathbf{a}$, which is not possible in case of a valid chase step. □

Let $\mathcal{G}$ be a GDatalog[Δ] program, I be an input for $\mathcal{G}$, and T be a chase tree. The next theorem shows that there is a bijective correspondence between the maximal paths of T and the possible outcomes of $\mathcal{G}$. This gives us a crucial tool to construct the probability measure of Theorem 3.8. First, some notation is needed. 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 4.3. *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 $\Omega_{\mathcal{G}}(I)$.*

Proof. We first prove that every $\cup P$ is in $\Omega_{\mathcal{G}}(I)$. Let $P \in$ *paths*(T) be given. We need to show that $\cup P \in \Omega_{\mathcal{G}}(I)$. By definition, it is the case that every distributional fact of $\cup P$ has a nonzero probability. It is also clear that $\cup P$ is consistent, due to the fairness property of T. Hence, it suffices to prove that $\cup P$ is a *minimal* solution; that is, no proper subset of $\cup P$ is a solution. So, let K be a strict subset of $\cup P$ and suppose, by way of contraction, that K is also a solution. Let (J, J') be the first edge in P such that $\cup P$ contains a fact that is not in K. Now, consider the chase step that leads from J to J'. Let f be the unique fact in $J' \setminus J$. Then $J \subseteq K$ and $f \in J' \setminus K$. The selected rule τ in this step cannot be deterministic, or otherwise K must contain f as well. Hence, it is a distributional rule, and f has the form $R_i^{\delta}(\mathbf{a}|\mathbf{p})$. But then K satisfies this rule, and hence, K must include a fact $f' = R_i^{\delta}(\mathbf{a}'|\mathbf{p})$, where $\mathbf{a}'$ differs from $\mathbf{a}$ only in the ith element. And since some node in $\cup P$ contains both f and f', we get a violation of the functional dependency of Proposition 4.2. Hence, a contraction.

Next, we prove that every possible outcome J in $\Omega_{\mathcal{G}}(I)$ is equal to $\cup P$ for some $P \in$ *paths*(T). Let such J be given. We build the path P inductively, as follows. We start with the root, and upon every node v we select the next edge to be one that leads to a subset K of J; note that K must exist since J resolves the rule violated in $L(v)$ by some fact, and that fact must be in one of the children of v. Now, $\cup P$ is consistent since T is fair, and $\cup P \subseteq J$ by construction. And since J is a minimal solution, we get that $\cup P$ is in fact equal to J.

Finally, we need to prove that if $\cup P_1 = \cup P_2$, then $P_1 = P_2$. We will prove the contrapositive statement. Suppose that $P_1, P_2 \in$ *paths*(T) are such that $P_1 \neq P_2$. The two paths agree on the root. Let J be the first node in the paths such that the two paths disagree on the outgoing edge of J. Suppose that P_1 has the edge from J to J_1 and P_2 has an edge from J to J_2. Then $J_1 \cup J_2$ has a pair of facts that violate the functional dependency of Proposition 4.2, and in particular, $J_1 \nsubseteq \cup P_2$. We conclude that $\cup P_1 \neq \cup P_2$, as required. □

4.1.1 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 4.3, we can do that by defining a probability measure over *paths*(T). A random path in *paths*(T) can be viewed as an infinite *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* (Ash and Doleans-Dade 2000). 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. The following theorem is a special case of a classic result on Markov chains (cf. Ash and Doleans-Dade (2000, Section 2.7)), stating that if every step is a well-defined probability measure space, then the infinite chain defines a probability measure space that is unique, defined by the cylinders and their probabilities. 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.

Theorem 4.4. *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 conditions:*

(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 4.3 and 4.4 suggest the following definition.

Definition 4.5 (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 4.4. 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 $\cup P$.

Our main result for this section is Theorem 4.7, which states that the probability measure space represented by a chase tree is independent of the specific chase tree of choice. In other words, the semantics is robust to the chase order, as long as fairness is guaranteed. To state and prove this result, we need some notation and a lemma. Let $\mathcal{G}$ be a GDatalog[Δ] program, let I be an input for $\mathcal{G}$, let T be a chase tree, and let v be a node of T. We denote by $\cup C_v^T$ the set $\{\cup P \mid P \in C_v^T\}$.

Lemma 4.6. *Let $\mathcal{G}$ be a GDatalog[Δ] program, let I be an input for $\mathcal{G}$, and let T be a fair chase tree. Let v be a node of T and $F = L(v)$. Then $\cup C_v^T = \Omega_{\mathcal{G}}^{F\subseteq}(I)$; that is, $\cup C_v^T$ is the set $\{J \in \Omega_{\mathcal{G}}(I) \mid L(v) \subseteq J\}$.*

Proof. From Theorem 4.3, it follows that every fact set in $\cup C_v^T$ is a possible outcome; since each such possible outcome contains $L(v)$, we have that

$$\cup C_v^T \subseteq \{J \in \Omega_{\mathcal{G}}(I) \mid L(v) \subseteq J\}\,.$$

Theorem 4.3 also implies that every possible outcome is equal to $\cup P$ for some maximal path P in $paths(T)$. If such P is not in C_v^T, then it branches out from some ancestor of v, say, u. Proposition 4.2 implies that $\cup P$ and $L(v)$ satisfy a functional dependency, for the distributional fact associated with u, but with different values. It does follow that P does not contain $L(v)$. Therefore, we have the reverse containment:

$$\{J \in \Omega_{\mathcal{G}}(I) \mid L(v) \subseteq J\} \subseteq \cup C_v^T\,.$$

This completes the proof. □

We can now prove our result.

Theorem 4.7. *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 $\mu_T = \mu_{T'}$.*

Proof. Let $\mu_T = (\Omega, \mathcal{F}, \pi)$ and $\mu_{T'} = (\Omega', \mathcal{F}', \pi')$. We need to prove that $\Omega = \Omega'$, $\mathcal{F} = \mathcal{F}'$, and $\pi = \pi'$. We have $\Omega = \Omega'$ due to Theorem 4.3. To prove that $\mathcal{F} = \mathcal{F}'$, it suffices to prove that every $\cup C_v^T$ is in $\mathcal{F}'$ and every $\cup C_{v'}^{T'}$ is in $\mathcal{F}$ (since both σ-algebras are generated by the cylinders). And due to symmetry, it suffices to prove that $C_{v'}^{T'}$ is in $\mathcal{F}$. So, let v' be a node of T'. Recall that $L(v')$ is a set of facts. Due to Lemma 4.6, we have that $\cup C_{v'}^{T'}$ is precisely the set of all possible outcomes J in $\Omega_{\mathcal{G}}(I)$ such that $L(v') \subseteq J$. Let U be the set of all nodes u of T with $L(v') \subseteq L(u)$. Then, due to Theorem 4.3, we have that $\cup C_{v'}^{T'} = \cup_{u \in U}(\cup C_u^T)$. Observe that U is countable, since T has only a countable number of nodes (as every node is identified by a finite path from the root). Moreover, $(\Omega, \mathcal{F})$ is closed under countable unions, and therefore, $\cup_{u \in U}(\cup C_u^T)$ is in $\mathcal{F}$.

It remains to prove that $\pi = \pi'$. By now we know that the σ-algebras $(\Omega, \mathcal{F})$ and $(\Omega', \mathcal{F}')$ are the same. Due to Theorem 4.3, every measure space over $(\Omega, \mathcal{F})$ can be translated into a measure space over the cylinder algebra of T and T'. So, due to the uniqueness property of Theorem 4.4, it suffices to prove that every $\cup C_{v'}^{T'}$ has the same probability in μ_T and $\mu_{T'}$. That is, $\pi(\cup C_{v'}^{T'}) = \mathbf{P}(L(v'))$. We do so next. We assume that v' is not the root of T', or otherwise the claim is straightforward. Let U be the set of all the nodes u in T with the property that $L(v') \subseteq L(u)$ but $L(v') \nsubseteq L(p)$ for the parent p of u. Due to Lemma 4.6, we have the following:

$$\pi(\cup C_{v'}^{T'}) = \sum_{u \in U} \mathbf{P}(L(u)). \tag{3}$$

Let E be the set of all the edges (v_1, u_1) of T such that $L(u_1) \setminus L(v_1)$ consists of a fact in $L(v')$. Let Q be the set of all the paths from the root of T to nodes in U. Due to Proposition 4.2, we have that every two paths P_1 and P_2 in Q and edges (v_1, u_1) and (v_2, u_2) in P_1 and P_2, respectively, if both edges are in E and $v_1 = v_2$, then $u_1 = u_2$. Let T'' be the tree that is obtained from T by considering every edge (v_1, u_1) in E, changing its weight to 1, and changing the weights of the remaining (v_1, u_1') emanating from v_1 to 0. Then we have the following for every node $u \in U$:

$$\mathbf{P}(L(u)) = w_{T''}(u) \cdot \mathbf{P}(L(v')), \tag{4}$$

where $w_{T''}(u)$ is the product of the weights along the path from the root of T'' to u. Combining Equations (3) and (4), we get the following:

$$\pi(\cup C_{v'}^{T'}) = \mathbf{P}(L(v')) \cdot \sum_{u \in U} w_{T''}(u).$$

Let $s = \sum_{u \in U} w_{T''}(u)$. We need to prove that $s = 1$. Observe that s is the probability of visiting a node of U in a random walk over T'' (with the probabilities defined by the weights). Equivalently, s is the probability that the random walk over T'' eventually sees all of the facts in v'. But due to the construction of T'', every rule violation that arises due to facts in both $L(v')$ and any node of T'' is deterministically resolved exactly as in $L(v')$. Moreover, since $L(v')$ is obtained from a chase derivation (i.e., $L(v')$ is a derivation set), solving all such rules repeatedly results in the containment of $L(v')$. Finally, since T'' is fair (because T is fair), we get that every random walk over T'' eventually sees all of the facts in $L(v')$. Hence, $s = 1$, as claimed. □

4.2 Proof of Theorem 3.8

We can now prove Theorem 3.8. Let $\mathcal{G}$ be a GDatalog[Δ] program, let I be an input for $\mathcal{G}$, and let T be a fair chase tree for I w.r.t. $\mathcal{G}$. Let $\mu_T = (\Omega_{\mathcal{G}}(I), \mathcal{F}_T, \pi_T)$ be the probability measure on $\Omega_{\mathcal{G}}(I)$ associated with T, as defined in Definition 4.5.

Lemma 4.8. *The σ-algebra $(\Omega_{\mathcal{G}}(I), \mathcal{F}_T)$ is generated by the sets of the form $\Omega_{\mathcal{G}}^{F\subseteq}(I)$, where F is finite.*

Proof. Let $(\Omega_{\mathcal{G}}(I), \mathcal{F})$ be the σ-algebra generated from the sets $\Omega_{\mathcal{G}}^{F\subseteq}(I)$. We will show that every $\Omega_{\mathcal{G}}^{F\subseteq}(I)$ is in $\mathcal{F}_T$, and that every $\cup C_v^T$ is in $\mathcal{F}$. The second claim is due to Lemma 4.6, so we will prove the first. So, let $\Omega_{\mathcal{G}}^{F\subseteq}(I)$ be given. Due to Lemma 4.6, the set $\Omega_{\mathcal{G}}^{F\subseteq}(I)$ is the countable union $\cup_{u \in U}(\cup C_u^T)$, where U is the set of all the nodes u such that $F \subseteq L(u)$. Hence, $\Omega_{\mathcal{G}}^{F\subseteq}(I) \in \mathcal{F}_T$. □

Lemma 4.9. *For every derivation set F we have $\pi_T(\Omega_{\mathcal{G}}^{F\subseteq}(I)) = \mathbf{P}(F)$.*

Proof. Let F be a derivation set. Due to Theorem 4.7, it suffices to prove that for *some* chase tree T', it is the case that $\pi_{T'}(\Omega_{\mathcal{G}}^{F\subseteq}(I)) = \mathbf{P}(F)$. But since F is a derivation set, we can craft a chase tree

T' that has a node v with $L(v) = F$. Then we have that $\pi_{T'}(\Omega_{\mathcal{G}}^{F\subseteq}(I))$ is the product of the weights along the path to v, which is exactly $\mathbf{P}(F)$. □

LEMMA 4.10. *Let $\mu = (\Omega, \mathcal{F}, \pi)$ be any probability space that satisfies the three conditions of Theorem 3.8. Then $\mu = \mu_T$.*

PROOF. Let $\mu_T = (\Omega_{\mathcal{G}}(I), \mathcal{F}_T, \pi_T)$. Due to Lemma 4.8, we have that $\mathcal{F} = \mathcal{F}_T$. So it is left to prove that $\pi = \pi_T$. Due to Lemmas 4.9 and 4.6, π agrees with π_T. Due to Theorems 4.3 and 4.4, we get that π must be equal to π_T due to the uniqueness of π_T. □

The previous lemmas show that $\mu_T = (\Omega_{\mathcal{G}}(I), \mathcal{F}_T, \pi_T)$ is a probability measure space that satisfies Equations (1) through (3) of Theorem 3.8, and moreover, that no other probability measure space satisfies Equations (1) through (3).

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. After defining PPDL, we prove that the probabilistic semantics of PPDL programs is preserved under *logical equivalence* (Theorem 5.5). To do so, we define two concepts of *equivalence* of PPDL programs: *first-order equivalence* (the programs are equivalent when viewed as ordinary first-order theories) and *semantic equivalence* (defining the same probability measure spaces). Finally, we discuss the decidability of the two properties for generative Datalog (Theorem 5.6).

Definition 5.1 (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 $\mathcal{E} \cup \mathcal{I}$.[4]

Example 5.2. Consider again Example 3.2. 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) ObservedHAlarm$(h) \rightarrow$ Alarm(h)
(2) ObservedBAlarm$(b) \rightarrow$ Alarm(b)

We use right (in contrast to left) arrows to distinguish constraints from ordinary Datalog rules. A possible outcome J of an input instance I satisfies these constraints if J contains 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 5.3. 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:

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

(1) $\Omega_{\mathcal{P}}(I) = \{J \in \Omega_{\mathcal{G}}(I) \mid J \models \Phi\}$,
(2) $\mathcal{F}_{\mathcal{P}} = \{S \cap \Omega_{\mathcal{P}}(I) \mid S \in \mathcal{F}_{\mathcal{G}}\}$, and
(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 5.4. Continuing Example 5.2, the semantics of this program is the posterior probability distribution that is obtained from the prior of Example 3.2, 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 Earthquake(Napa, 1).

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, 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. We make this statement formal in Theorem 5.5.

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 is used with different numbers of arguments, such as Flip in Figure 2, then 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. The following theorem states that FO-equivalence implies semantic equivalence. Hence, rewriting a program in a way that preserves logical equivalence is safe, as the semantics is preserved.

Theorem 5.5. *If two PPDL[Δ] programs are FO-equivalent, then they are semantically equivalent (but not necessarily vice versa).*

Proof. For simplicity, we will first restrict attention to constraint-free programs. Let $\mathcal{G}_1 = (\mathcal{E}, \mathcal{I}, \Theta_1)$ and $\mathcal{G}_2 = (\mathcal{E}, \mathcal{I}, \Theta_2)$ be FO-equivalent GDatalog[Δ] programs, and let I be any input instance. We will show that $\Omega_{\mathcal{G}_1}(I) = \Omega_{\mathcal{G}_2}(I)$. It then follows from Theorem 3.8 that $\mu_{\mathcal{G}_1,I}$ and $\mu_{\mathcal{G}_2,I}$ coincide.

Recall that a *model* of an arbitrary first-order theory Θ in a signature $\{R_1, \ldots, R_n, c_1, \ldots, c_m, f_1, \ldots, f_k\}$ is a structure $M = (Dom, R_1^M, \ldots, R_n^M, c_1^M, \ldots, c_m^M, f_1^M, \ldots, f_k^M)$, where Dom is a set called the *domain* of M, each R_i^M is a relation over Dom of appropriate arity, each c_i^M is an element of Dom, and each f_i^M is a function on Dom of appropriate arity, such that $M \models \Theta$; that is, all sentences in Θ are satisfied in M. Recall that all constants that may occur in PPDL[Δ] program are real numbers. We say that a structure is *real-valued* if $Dom = \mathbb{R}$ and each constant symbol is a real number denoting itself. We say that a structure M is a *relationally minimal model* of a first-order theory Γ if M is a model of Γ and no structure obtained from M by dropping one or more tuples from relations is a model of Γ.

We denote by $\Omega'_{\mathcal{G}}(I)$ the set of relationally minimal real-valued models of $\mathcal{G} \cup I$ (where I is viewed as a theory as well, more specifically, a set of ground atomic facts). If two first-order theories are FO-equivalent, a fortiori, they have the same relationally minimal real-valued models. Since we have assumed that $\mathcal{G}_1$ and $\mathcal{G}_2$ are FO-equivalent, we have that $\Omega'_{\mathcal{G}_1}(I) = \Omega'_{\mathcal{G}_2}(I)$.

We will now establish a correspondence between models $M \in \Omega'_{\mathcal{G}_i}(I)$ and possible outcomes $J \in \Omega_{\mathcal{G}}(I)$. To formulate this correspondence, we need one more definition, namely , that of an *active term*. We denote by RelDiag(M) the *relational diagram* of M, that is, the set of all ground relational facts $R_i(\vec{a})$ that are true in M. We say that a ground term of the form $f(a_1, \ldots, a_n)$ is *active* in a model M relative to a first-order theory Γ, if $\Gamma \cup$ RelDiag(M) entails an atomic formula containing the term $f(a_1, \ldots, a_n)$. For example, if $\Gamma = \{\forall x P(x) \rightarrow Q(f(x))\}$ and $M \models P(1)$, then the term $f(1)$ is active in M, because $\Gamma \cup \{P(1)\} \models Q(f(1))$. The intuition behind this definition is as follows: a first-order structure must, by definition, always interpret a function symbol by a total function (defined on all inputs). When a term $f(a_1, \ldots, a_n) \in M$ is *active* with respect to Γ, it means that it actually plays an active role in the satisfaction of Γ in M.

For each $M \in \Omega'_{\mathcal{G}}(I)$, let $\bar{M}$ be the instance consisting of all relational facts (i.e., facts of the form $R(\mathbf{a})$ of M), and all relational facts of the form Result$^f(\mathbf{a}, b)$, where $f^M(\mathbf{a}) = b$ and $f(\mathbf{a})$ is active in M w.r.t. $\mathcal{G}$. Then,

(1) for each $M \in \Omega'_{\mathcal{G}}(I)$, we have that $\bar{M} \in \Omega_{\mathcal{G}}(I)$; and
(2) each member of $\Omega_{\mathcal{G}}(I)$ coincides with $\bar{M}$ for some $M \in \Omega'_{\mathcal{G}}(I)$.

The argument for (2) uses Proposition 3.6.

In particular, it follows from (1) and (2) that $\Omega'_{\mathcal{G}}(I)$ uniquely determines $\Omega_{\mathcal{G}}(I)$. Since we have already established that $\Omega'_{\mathcal{G}_1}(I) = \Omega'_{\mathcal{G}_2}$, we conclude that $\Omega_{\mathcal{G}_1}(I) = \Omega_{\mathcal{G}_2}(I)$. Since I was chosen arbitrarily, we conclude that $\mathcal{G}_1$ and $\mathcal{G}_2$ are semantically equivalent.

The argument immediately extends to the case of PPDL[Δ] programs with constraints: let $\mathcal{P}_1 = (\mathcal{E}, \mathcal{I}, \Theta_1, \Phi_1)$ and $\mathcal{P}_2 = (\mathcal{E}, \mathcal{I}, \Theta_2, \Phi_2)$ be FO-equivalent PPDL[Δ] programs. Then, in particular, $\mathcal{G}_1 = (\mathcal{E}, \mathcal{I}, \Theta_1)$ and $\mathcal{G}_2 = (\mathcal{E}, \mathcal{I}, \Theta_2)$ are FO-equivalent GDatalog[Δ] programs, and hence, as we have shown, for all input instances I, $\mu_{\mathcal{G}_1,I}$ and $\mu_{\mathcal{G}_2,I}$ coincide. Furthermore, Φ_1 and Φ_2 are FO-equivalent. It then follows by the semantics of PPDL constraints (and for legal input instances I) that $\mu_{\mathcal{P}_1,I}$ and $\mu_{\mathcal{P}_2,I}$ coincide as well. In other words, P_1 and P_2 are semantically equivalent.

Semantic equivalence does not imply FO-equivalence. Indeed, let P, Q be IDB predicates, and consider the program consisting of the rule $Q(x) \leftarrow P(x)$. Since there are no rules that derive IDB-facts from EDB-facts, this program is semantically equivalent to the program without any rules. Nevertheless, these two programs are not FO-equivalent. Intuitively, semantic equivalence is only concerned with minimal models (since a minimality requirement is built into the notion of a possible outcome), while FO-equivalence is concerned with arbitrary models. In addition to this, first-order equivalence is oblivious to facts about the specific probability distributions in Δ. □

The following theorem discusses the decidability of two types of equivalences for GDatalog[Δ] and shows that FO-equivalence is decidable for weakly acyclic programs, while semantic equivalence is not.

Theorem 5.6. *First-order equivalence is decidable for weakly acyclic GDatalog[Δ] programs. Semantic equivalence is undecidable for weakly acyclic GDatalog[Δ] programs (in fact, even for $\Delta = \emptyset$).*

Proof. The decidability is proved using the following well-known technique: for each rule of one of the two programs, we take the canonical instance of the right-hand side of the rule, chase it with the other program, and test that the left-hand side of the rule is in the result of the chase. In this way, we can test containment in both directions (i.e., equivalence of the two programs). Before spelling out the procedure, we illustrate it with an example. Consider the following two GDatalog[Δ] programs:

— $\mathcal{G}_1$ is defined by the following rules:
(1) $Q(x) \leftarrow P(x)$; (2) $P(x) \leftarrow Q(x)$; (3) $R(\delta[x]) \leftarrow P(x)$
— $\mathcal{G}_2$ is defined by the following rules:
(1) $Q(x) \leftarrow P(x)$; (2) $P(x) \leftarrow Q(x)$; (3) $R(\delta[x]) \leftarrow Q(x)$

To test that $\mathcal{G}_1$ and $\mathcal{G}_2$ are equivalent, we test that $\mathcal{G}_1$ logically implies each rule of $\mathcal{G}_2$, and, conversely, $\mathcal{G}_2$ logically implies each rule of $\mathcal{G}_1$. Each can be tested by chasing the right-hand side of the rule in question and verifying that the left-hand side belongs to the result of the chase. For example, to show that $\mathcal{G}_1$ logically implies the third rule of $\mathcal{G}_2$, we consider the canonical instance of its right-hand side (i.e., $\{Q(a)\}$) and chase it with the rules of $\mathcal{G}_1$, obtaining the chase result $\{Q(a), P(a), R(\delta[a])\}$. Indeed, it can be confirmed that the left-hand side of the rule in question is satisfied (under the same assignment of values to variables, i.e., $R(\delta[a])$) in the chase result.

The general procedure for testing that two weakly acyclic GDatalog[Δ] programs $\mathcal{G}_1$ and $\mathcal{G}_2$ are FO-equivalent is as follows: for each rule r of $\mathcal{G}_1$, we take a canonical instance of the right-hand side of r (replacing each variable by a fresh, distinct constant) and chase it with the rules of $\mathcal{G}_2$. Note that, by weak acyclicity, this chase is guaranteed to terminate and yield a finite instance. We then test that the canonical instance of the left-hand side of r (under the same replacement of variables by constants) is contained in the result of this chase. Likewise, for each rule r of $\mathcal{G}_2$ we take a canonical instance of the right-hand side of r (replacing each variable by a fresh, distinct constant), chase it with the rules of $\mathcal{G}_1$, and test that the canonical instance of the left-hand side of r (under the same replacement of variables by constants) is contained in the result of this chase. If each of these tests succeeds, then $\mathcal{G}_1$ and $\mathcal{G}_2$ must be FO-equivalent. If not, then the canonical instance of the failing rule provides us with a counterexample to the FO-equivalence of $\mathcal{G}_1$ and $\mathcal{G}_2$.

The undecidability claim follows immediately from the undecidability of equivalence of Datalog programs (Shmueli 1993). □

Remark 5.7. We remind the reader that, in the programs $\mathcal{G}_1$ and $\mathcal{G}_2$ used in the last proof, "$R(\delta[x])$" uses our sugared syntax, and it stands for the formal syntax defined in Section 3, which in this case would be "$R(\delta[\![x;]\!])$." This is an important point, since the correctness of Theorem 5.6 is based on the fact that we define FO-equivalence by reference to the unsugared program.

Remark 5.8. The PPDL[Δ] programs given in Figures 2 and 3 are not semantically equivalent: once the syntactic sugar in the second program is expanded, the fourth rule of this program may generate R_4^{Flip} facts, whereas no possible solution for the first program contains any R_4^{Flip} fact. Nevertheless, the two programs are equivalent in a weaker sense. To make this precise, we define another probability space, $\check{\mu}_{\mathcal{G},I}$. Let $\mathcal{P} = (\mathcal{E}, \mathcal{I}, \Theta, \Phi)$, let I be an input instance over $\mathcal{E}$, and let $J \in \Omega_{\mathcal{P}}(I)$ be a possible outcome for I w.r.t. $\mathcal{P}$. Recall that J is an instance over the schema $\mathcal{E} \cup \mathcal{I}^{\Delta}$. We denote by $\check{J}$ the instance over schema $\mathcal{E} \cup \mathcal{I}$ that is obtained from J by dropping all facts using relation symbols from $\mathcal{I}^{\Delta} \setminus \mathcal{I}$. In other words, $(\check{\cdot})$ is the natural "forgetful" map that transforms $\mathcal{E} \cup \mathcal{I}^{\Delta}$-instances to $\mathcal{E} \cup \mathcal{I}$-instances. We define $\check{\mu}_{\mathcal{P},I}$ to be the image of the measure space $\mu_{\mathcal{P},I}$ under the map $(\check{\cdot})$ (a standard topological operation). Then $(\check{\cdot})$ is a σ-algebra homomorphism from $\mu_{\mathcal{P},I}$ to $\check{\mu}_{\mathcal{P},I}$. Intuitively, all that we have done is "forget" all information regarding distributional facts. We say that two PPDL[Δ] programs, $\mathcal{P}_1, \mathcal{P}_2$, are *weakly semantically equivalent* if, for all instances I, the probability spaces $\check{\mu}_{\mathcal{G}_1,I}$ and $\check{\mu}_{\mathcal{G}_2,I}$ coincide.

Semantic equivalence implies weak semantic equivalence. Weak equivalence does not imply strong equivalence: the program consisting of the rules $P(x,0) \leftarrow Q(x)$, $P(x,1) \leftarrow Q(x)$, and $P(x, \mathsf{Flip}[0.5; x]) \leftarrow Q(x)$ is weakly equivalent, but not strongly equivalent, to the same program in which the third rule is dropped. It can be shown that the PPDL[Δ] programs given in Figure 2 and Figure 3 are indeed weakly semantically equivalent.

For PPDL[Δ] programs with $\Delta = \emptyset$, semantic equivalence and weak semantic equivalence coincide. Therefore, Theorem 5.6 implies that weak semantic equivalence of weakly acyclic PPDL[Δ] programs is undecidable as well.

6 PPDL EXAMPLES

To further illustrate PPDL, we now show two examples of translations thereto. The first is a translation from a *Markov Logic Network* (MLN) (Domingos and Lowd 2009), and the second from a *stochastic context-free grammar.*

6.1 Markov Logic Networks in PPDL

Jha and Suciu (2012) showed how MLN *inference* can be reduced to inference in tuple-independent probabilistic databases. More precisely, they considered the task of computing the probability of a Boolean query over the probability space defined by an MLN. They showed that this probability is a function of the probabilities of a few Boolean queries over a tuple-independent probabilistic database. Since the latter can be represented in GDatalog[Δ], assuming Δ contains the distribution Flip, then the same reduction can be done for GDatalog[Δ] and PPDL[Δ]. In this section, we show a stronger result, where the application of the PPDL[Δ] program to the input instance induces *the exact same* probability distribution as the MLN. Next, we introduce the MLN formalism, and then show how PPDL can capture it fairly simply.

6.1.1 Markov Logic Networks. We view an MLN as a pair of a program consisting of *constraints* and of *domains* of attributes.

Definition 6.1 (MLN Program). An *MLN program* is a tuple $(\mathcal{T}, \Gamma, \Psi, W)$, where:

— $\mathcal{T}$ is a relational schema.
— Ψ and Γ are sets of first-order-logic formulas over $\mathcal{T}$ such that every free variable occurs in at least one atomic formula (over a relation symbol of $\mathcal{T}$). Formulas in Ψ are *hard constraints* and formulas in Γ are *soft constraints*.
— $W : \Gamma \to \mathbb{R}$ is the *weight function* that associates a weight with every soft constraint.

Let $Q = (\mathcal{T}, \Psi, \Gamma, W)$ be an MLN program. An *attribute* of Q is a pair (R, i) where R is a relation symbol of $\mathcal{T}$ and $1 \leq i \leq \text{arity}(R)$. The *domain schema* of Q is the relational schema that consists of a unary relation symbol $R_{[i]}$ for every attribute (R, i) of Q. An *MLN* is a pair (Q, D) where Q is an MLN program and D is an instance over the domain schema of Q. For short, we call D a *domain instance* for Q.

Example 6.2. For illustration, we use a simplified version of a bibliography resolution task (Singla and Domingos 2006), where we have a collection of bibliography entries from various resources, where common references may be represented differently (due to format differences and/or typos). This task is one of the popular instances of the general problem of *entity resolution.* In our example, we construct an MLN program $Q = (\mathcal{T}, \Psi, \Gamma, W)$, where $\mathcal{T}$ consists of the following relation symbols:

— InBib(a, b): entry b mentions author name a;
— SimilarTitle(b_1, b_2): entries b_1 and b_2 have a *similar* title (by some similarity test);
— SimilarName(a_1, a_2): author names a_1 and a_2 are similar (by some other test);
— SameAuthor(a_1, a_2): author names a_1 and a_2 refer to the same real-life author;
— SameBib(b_1, b_2): entries b_1 and b_2 refer to the same resource (e.g., article).

In an application of the program, the relations InBib, SimilarTitle, and SimilarName are given as observations, and the goal is to infer SameAuthor and SameBib (or just one of them). A typical domain instance D for Q associates with each b attribute the set of bibliography entries, and with each a attribute the set of author names in those entries. Hence, in such an instance, the unary relations $\text{InBib}_{[2]}$, $\text{SimilarTitle}_{[1]}$, $\text{SimilarTitle}_{[2]}$, $\text{SameBib}_{[1]}$, and $\text{SameBib}_{[2]}$ are the same—the list of given bibliography entries. Similarly, $\text{InBib}_{[1]}$, $\text{SimilarName}_{[1]}$, $\text{SimilarName}_{[2]}$, $\text{SameAuthor}_{[1]}$, and $\text{SameAuthor}_{[2]}$ are the same—the list of author names.

The set Ψ consists of four hard constraints that require SameAuthor and SameBib to be equivalence relations:

(**Hard1**) $\text{SameAuthor}(a, a)$
(**Hard2**) $\text{SameAuthor}(a_1, a_2) \rightarrow \text{SameAuthor}(a_2, a_1)$
(**Hard3**) $\text{SameAuthor}(a_1, a_2) \wedge \text{SameAuthor}(a_2, a_3) \rightarrow \text{SameAuthor}(a_1, a_3)$
(**Hard4**) $\text{SameBib}(b, b)$
(**Hard5**) $\text{SameBib}(b_1, b_2) \rightarrow \text{SameBib}(b_2, b_1)$
(**Hard6**) $\text{SameBib}(b_1, b_2) \wedge \text{SameBib}(b_2, b_3) \rightarrow \text{SameBib}(b_1, b_3)$

Finally, the set Γ consists of the following soft rules, where each rule γ is preceded by its weight $W(\gamma)$:

(**soft1**) $3 : \text{SameAuthor}(a_1, a_2) \rightarrow \text{SimilarName}(a_1, a_2)$
(**soft2**) $1 : \text{SimilarName}(a_1, a_2) \rightarrow \text{SameAuthor}(a_1, a_2)$
(**soft3**) $3 : \text{SameBib}(b_1, b_2) \rightarrow \text{SimilarTitle}(b_1, b_2)$
(**soft4**) $1 : \text{SimilarTitle}(b_1, b_2) \rightarrow \text{SameBib}(b_1, b_2)$
(**soft5**) $-5 : \text{SameBib}(b_1, b_2) \wedge \text{InBib}(a_1, b_1) \wedge \forall a_2[\text{InBib}(a_2, b_2) \rightarrow \neg\text{SameAuthor}(a_1, a_2)]$

The first four rules reward occurrences of consistency between similarity and identity, whereas the fifth rule penalizes a world for every two entries b_1 and b_2 and author name a_1 such that a_1 is in b_1 but no author name in b_2 is equal to a_1.

Let (Q, D) be an MLN where $Q = (\mathcal{T}, \Psi, \Gamma, W)$. A *grounding* of a (hard or soft) constraint φ of Q is obtained from φ by replacing every free variable with a constant from a suitable domain; that is, if a variable occurs in an attribute (R, i), then it is replaced with a constant from $R^D_{[i]}$. Denote by $\text{grd}(\varphi, D)$ the set of all groundings of φ. Then $\text{grd}(\varphi, D)$ is a set of Boolean propositional formulas (without free variables) over $\mathcal{T}$. A *possible world* of (Q, D) is an instance I of $\mathcal{T}$ that satisfies the following conditions:

(1) On every attribute (R, i), the instance I has only constants from $R^D_{[i]}$.
(2) $I \models g$ for every hard constraint $\psi \in \Psi$ and grounding $g \in \text{grd}(\psi)$.

We denote by $\mathbf{PW}(Q, D)$ the set of possible worlds of (Q, D). The *weight* $W(I)$ of a possible world I is given by

$$\text{weight}(I) \overset{\text{def}}{=} \exp\left(\sum_{\gamma \in \Gamma} \sum_{\substack{g \in \text{grd}(\gamma) \\ I \models g}} W(\gamma)\right). \tag{5}$$

An MLN (Q, D) defines a discrete probability space (Ω, π), where $\Omega = \mathbf{PW}(Q, D)$ and π is given by $\pi(I) = \text{weight}(I)/Z$, where the normalization factor Z, also called the *partition function*, is equal to $\sum_{J \in \mathbf{PW}(Q,D)} \text{weight}(J)$. This probability space is denoted by $\Pi_{Q,D}$.

6.1.2 Translation into PPDL. We now turn to PPDL and show that MLN can be captured by PPDL. In order to obtain the same probability distribution as the given MLN, we need to eliminate the intermediate (helper) predicates from the possible worlds, and we do so in a straightforward way by *projecting* over the relations of interest. A formal definition follows.

Let $\mathcal{P} = (\mathcal{E}, \mathcal{I}, \Theta, \Phi)$ be a PPDL[Δ] program, and let I be an instance over $\mathcal{E}$. Let $\mathcal{I}'$ be a sub-schema of $\mathcal{I}$; that is, $\mathcal{I}'$ is obtained from $\mathcal{I}$ by removing some of the relation symbols. For a possible outcome J, we denote by $J[\mathcal{I}']$ the restriction of J to the relations of $\mathcal{I}'$ (i.e., $J[\mathcal{I}']$ consists of the relations R^J where R belongs to $\mathcal{I}'$). Suppose that $\mu_{\mathcal{P},I}$ is a discrete probability space (i.e., the sample space is countable and every set of samples is measurable), and let us denote it by (Ω, π). The *projection* of $\mu_{\mathcal{P},I}$ to $\mathcal{I}'$ is the probability space that is obtained from $\mu_{\mathcal{P},I}$ by replacing every J with $J[\mathcal{I}']$ and assigning to $J[\mathcal{I}']$ its marginal probability. More formally, the projection of $\mu_{\mathcal{P},I}$ to $\mathcal{I}'$ is the discrete probability space (Ω', π') where:

— $\Omega' \overset{\text{def}}{=} \{J[\mathcal{I}'] \mid J \in \Omega\}$;
— $\pi'(J') \overset{\text{def}}{=} \sum_{J\in\Omega, J[\mathcal{I}]=J'} \pi(J)$.

To follow is the main result of this section, showing that every MLN program can be translated into a PPDL program, in the sense that the two define the same distribution when applied to the same domain. Moreover, the only numerical distribution function required for the PPDL program is Flip.

Theorem 6.3. *Let Δ be a set of distributions that includes* Flip. *For every MLN program $Q = (\mathcal{T}, \Psi, \Gamma, W)$ there exists a PPDL[Δ] program $\mathcal{P} = (\mathcal{E}, \mathcal{I}, \Theta, \Phi)$ with the following properties:*

(1) $\mathcal{E}$ is the domain schema of Q.
(2) $\mathcal{I}$ contains $\mathcal{T}$.
(3) Φ is a set of formulas in first-order logic.
(4) For every domain instance D of Q, the projection of $\mu_{\mathcal{P},D}$ to $\mathcal{T}$ is equal to $\Pi_{Q,D}$.

In the remainder of this section, we prove Theorem 6.3 by presenting a translation of an MLN program into a PPDL[Δ] program with $\Delta = \{\text{Flip}\}$. We fix an MLN program $Q = (\mathcal{T}, \Psi, \Gamma, W)$, and we denote by $\mathcal{E}$ the domain schema of Q. To define the PPDL[Δ] program $\mathcal{P} = (\mathcal{E}, \mathcal{I}, \Theta, \Phi)$, we then need to define $\mathcal{I}$, Θ, and Φ.

As a first step, we determine a random truth value for every possible fact over $\mathcal{T}$. For each relation symbol R in $\mathcal{T}$ with arity m, we add to Θ the following two rules:

$$\begin{aligned} R_{\mathcal{I}}(x_1, \ldots, x_m, \text{Flip}[0.5]) &\leftarrow R_{[1]}(x_1), \ldots, R_{[m]}(x_m) \\ R(x_1, \ldots, x_m) &\leftarrow R_{\mathcal{I}}(x_1, \ldots, x_m, 1). \end{aligned} \tag{6}$$

Next we handle the rules in Γ and Ψ. Let γ be a constraint in $\Gamma \cup \Psi$. We write γ as $\gamma(x_1, \ldots, x_k)$, where $x_1, \ldots, x_k$ are the variables that occur in γ. We consider three different cases:

(A) $\gamma \in \Psi$. We add γ as an observation (with a universal quantification over the domains when needed). We let $R^j_{[i_j]}$ be the relation in the domain schema $\mathcal{E}$ corresponding to x_j for $1 \le j \le k$:

$$R^1_{[i_1]}(x_1), \ldots, R^k_{[i_k]}(x_k) \rightarrow \gamma(x_1, \ldots, x_k). \tag{7}$$

(B) $\gamma \in \Gamma$ *and* $W(\gamma) \ge 0$. For every $j = 1, \ldots, k$, select an attribute of a relation where x_i occurs in γ, and let $R^j_{[i_j]}$ be the corresponding relation in the domain schema $\mathcal{E}$. We then add the following rules:

$$O_\gamma(x_1, \ldots, x_k, \text{Flip}[\exp(-W(\gamma))]) \leftarrow R^1_{[i_1]}(x_1), \ldots, R^k_{[i_k]}(x_k) \tag{8}$$

(1) $\mathrm{SameAuthor}_{\mathcal{I}}(a_1, a_2, \mathsf{Flip}[0.5]) \ \leftarrow\ \mathrm{AuthorName}(a_1), \mathrm{AuthorName}(a_2)$
(2) $\mathrm{SameAuthor}(a_1, a_2) \ \leftarrow\ \mathrm{SameAuthor}_{\mathcal{I}}(a_1, a_2, 1)$
(3) $\mathrm{SameBib}_{\mathcal{I}}(b_1, b_2, \mathsf{Flip}[0.5]) \ \leftarrow\ \mathrm{Bib}(b_1), \mathrm{Bib}(b_2)$
(4) $\mathrm{SameBib}(b_1, b_2) \ \leftarrow\ \mathrm{SameBib}_{\mathcal{I}}(b_1, b_2, 1)$
(5) $\mathrm{AuthorName}(a) \ \rightarrow\ \mathrm{SameAuthor}(a, a)$
(6) $\mathrm{SameAuthor}(a_1, a_2) \ \rightarrow\ \mathrm{SameAuthor}(a_2, a_1)$
(7) $\mathrm{SameAuthor}(a_1, a_2), \mathrm{SameAuthor}(a_2, a_3) \ \rightarrow\ \mathrm{SameAuthor}(a_1, a_3)$
(8) $\mathrm{BibEntry}(b) \ \rightarrow\ \mathrm{SameBib}(b, b)$
(9) $\mathrm{SameBib}(b_1, b_2) \ \rightarrow\ \mathrm{SameBib}(b_2, b_1)$
(10) $\mathrm{SameBib}(b_1, b_2), \mathrm{SameBib}(b_2, b_3) \ \rightarrow\ \mathrm{SameBib}(b_1, b_3)$
(11) $O_{\mathrm{soft1}}(a_1, a_2, \mathsf{Flip}[\exp(-3)]) \ \leftarrow\ \mathrm{AuthorName}(a_1), \mathrm{AuthorName}(a_2)$
(12) $O_{\mathrm{soft1}}(a_1, a_2, 0) \ \rightarrow\ [\mathrm{SameAuthor}(a_1, a_2) \rightarrow \mathrm{SimilarName}(a_1, a_2)]$
(13) $O_{\mathrm{soft2}}(a_1, a_2, \mathsf{Flip}[\exp(-1)]) \ \leftarrow\ \mathrm{AuthorName}(a_1), \mathrm{AuthorName}(a_2)$
(14) $O_{\mathrm{soft2}}(a_1, a_2, 0) \ \rightarrow\ [\mathrm{SimilarName}(a_1, a_2) \rightarrow \mathrm{SameAuthor}(a_1, a_2)]$
(15) $O_{\mathrm{soft3}}(b_1, b_2, \mathsf{Flip}[\exp(-3)]) \ \leftarrow\ \mathrm{Bib}(b_1), \mathrm{Bib}(b_2)$
(16) $O_{\mathrm{soft3}}(b_1, b_2, 0) \ \rightarrow\ [\mathrm{SameBib}(b_1, b_2) \rightarrow \mathrm{SimilarTitle}(b_1, b_2)]$
(17) $O_{\mathrm{soft4}}(b_1, b_2, \mathsf{Flip}[\exp(-1)]) \ \leftarrow\ \mathrm{Bib}(b_1), \mathrm{Bib}(b_2)$
(18) $O_{\mathrm{soft4}}(b_1, b_2, 0) \ \rightarrow\ [\mathrm{SimilarTitle}(b_1, b_2) \rightarrow \mathrm{SameBib}(b_1, b_2)]$
(19) $O_{\mathrm{soft5}}(a_1, b_1, b_2, \mathsf{Flip}[\exp(-5)]) \ \leftarrow\ \mathrm{AuthorMention}(a_1), \mathrm{Bib}(b_1), \mathrm{Bib}(b_2)$
(20) $O_{\mathrm{soft5}}(a_1, b_1, b_2, 0) \ \rightarrow$
$\mathrm{SameBib}(b_1, b_2) \vee \mathrm{InBib}(a_1, b_1) \vee \exists a_2[\mathrm{InBib}(a_2, b_2), \mathrm{SameAuthor}(a_1, a_2)]$

Fig. 8. PPDL translation of the MLN of Example 6.2.

$$O_\gamma(x_1, \ldots, x_k, 0) \ \rightarrow\ \gamma(x_1, \ldots, x_k). \tag{9}$$

Intuitively, the rule defines an "observer" for $\gamma(x_1, \ldots, x_k)$ that tosses a coin with probability $\exp(-W(\gamma))$ to 1, and we condition the result to be 1, unless $\gamma(x_1, \ldots, x_k)$ holds true. As we explain in the next section, this action effectively brings up the factor $\exp(W(\gamma))$ whenever $\gamma(x_1, \ldots, x_k)$ is true.

(C) $\gamma \in \Gamma$ *and* $W(\gamma) < 0$. We apply the construction of the previous cases, except that we replace γ with $\gamma' = \neg\gamma$ and define $W(\gamma') = -W(\gamma)$. The intuition is the following: as weights are normalized, multiplying the weight of each world that satisfies a grounding of γ by $\exp(W(\gamma))$ has the same effect as multiplying the weight of each world that violates that grounding by $\exp(-W(\gamma))$.

This completes our construction of the PPDL[Δ] program $\mathcal{P} = (\mathcal{E}, \mathcal{I}, \Theta, \Phi)$.

Example 6.4. Figure 8 depicts the PPDL program $\mathcal{P} = (\mathcal{E}, \mathcal{I}, \Theta, \Phi)$ of the MLN program $\mathcal{Q} = (\mathcal{T}, \Psi, \Gamma, W)$ that we introduced in Example 6.2. Rules (1) through (4) correspond to the construction of Equation (6); Rules (5) through (10) correspond to the hard constraints, where (for readability) we avoided quantification over the domains when it is natural to do so; Rules (11) through (18) correspond to the soft constraints with a positive weight; and Rules (19) through (20) correspond to the soft constraint with a negative weight. This PPDL program simplifies the general construction in two ways:

— Using the fact that domains are shared across attributes, we replaced all of the relational symbols $R_{[i]}$ with two unary relations: Bib for bibliography mentions, and AuthorName for author mentions. For example, in Rule (1) that corresponds to Equation (6) with R

being for SameAuthor, the body uses AuthorName twice instead of $\text{SameAuthor}_{[1]}$ and $\text{SameAuthor}_{[2]}$.

— Making the assumption that InBib, SimilarTitle, and SimilarName are always given as observations, we actually view them as EDBs in $\mathcal{E}$. This illustrates a general principle that, in our construction, only the inferred relations of the MLN need to be IDBs, and we can replace the observed relations by EDB relations.

Regarding Rule (20), the reader can verify that the conclusion is the negation of the corresponding soft rule (soft5), as required by our construction.

Our construction can be adapted to show that when the MLN is defined in a smaller fragment of first-order logic, the constraints of the obtained PPDL program also belong to the same fragment, provided that the fragment satisfies the suitable closure conditions.

6.1.3 Proof of Correctness. We now prove the correctness of the construction. Let D be a domain instance for Q, and let (Ω, π) be the probability space $\Pi_{Q,D}$. Recall that $Q = (\mathcal{T}, \Psi, \Gamma, W)$. Let (Ω', π') be the projection of $\mu_{\mathcal{P},D}$ to $\mathcal{T}$. We need to prove that (Ω, π) is equal to (Ω', π'). We show that for every instance J over $\mathcal{T}$ with attribute values from D, the probability of J is the same in both probability spaces. Since both are (discrete) probability spaces, it suffices to prove that the unnormalized probability of J in one is proportional (i.e., within a multiplicative factor that is independent of J) to the unnormalized probability of J in the other. In the remainder of this section, we fix such instance J.

If J violates a hard constraint γ in Ψ, then J is, by definition, not in Ω. But J cannot be in Ω' either, since violation of γ implies that Constraint (7) is violated as well. Conversely, if J satisfies γ, then Constraint (7) is satisfied. It thus follows that J is in Ω if and only if J is in Ω'. In the sequel, we assume that J satisfies Ψ and, hence, $J \in \Omega$ and $J \in \Omega'$. By definition, $\pi(J)$ is

$$\pi(J) \sim \exp\left(\sum_{\gamma \in \Gamma} \sum_{\substack{g \,\in\, \mathrm{grd}(\gamma) \\ J \models g}} W(\gamma)\right). \tag{10}$$

Consider the space of random chases of $(\mathcal{E}, \mathcal{I}, \Theta)$ over D. Recall that we are free to chase in whatever order we wish, and so, we assume that the chase follows the rules in the order we have defined them. (Observe that our PPDL program is weakly acyclic, and hence, every chase necessarily terminates.) Let (Ω_c, π_c) be the distribution over the resulting instances. Let E_J be the event that J is precisely the set of facts $R(a_1, \ldots, a_m)$ such that the chase produced $R_{\mathcal{I}}(x_1, \ldots, x_m, b)$ for $b = 1$ (as opposed to $b = 0$). Let E_O be the event that all of the groundings of the constraints defined in Equations (7) and (9) hold when the chase terminates. Then, we have the following:

$$\pi'(J) = \Pr(E_J \mid E_O) = \frac{\Pr(E_O \mid E_J) \cdot \Pr(E_J)}{\Pr(E_O)}.$$

The notation $\Pr(\cdot)$ applies to the probability space (Ω_c, π_c). Now, observe that $\Pr(E_O)$ is the same for all J. Moreover, $\Pr(E_J)$ is also the same for all J since Equation (6) assigns a uniform distribution to the assignment of 0/1 to the set of all $R_{\mathcal{I}}$s. We conclude the following:

$$\pi'(J) \sim \Pr(E_O \mid E_J).$$

Denote by Γ^+ the set $\{\gamma \in \Gamma \mid W(\gamma) \geq 0\}$ and by Γ^- the set $\{\gamma \in \Gamma \mid W(\gamma) < 0\}$. For a constraint γ and a grounding $g \in \mathrm{grd}(\gamma)$, let E_g denote the event that the corresponding grounding of Equation (7) or (9) is true. From the order of the chase, it follows that, given the event E_J (i.e., the truth assignment for $R_{\mathcal{I}}$s), the events E_g are independent across different γs. So, we have the following:

$$\Pr(E_O \mid E_J) = \left(\prod_{\gamma \in \Gamma^+} \prod_{g \in \mathrm{grd}(\gamma)} \Pr(E_g \mid E_J)\right) \times \left(\prod_{\gamma \in \Gamma^-} \prod_{g \in \mathrm{grd}(\gamma)} \Pr(E_{\neg g} \mid E_J)\right) \times \left(\prod_{\gamma \in \Psi} \prod_{g \in \mathrm{grd}(\gamma)} \Pr(E_g \mid E_J)\right).$$

We now process each of the three main factors. For a grounding g we denote by g_i the value that g assigns to the variable x_i. For $\gamma \in \Gamma^+$ and $g \in \mathrm{grd}(\gamma)$, the event E_g always holds if J satisfies g, due to Equation (9). If J violates g, then E_g holds with probability $\exp(-W(\gamma))$ due to Equation (8). We then have the following:

$$\left(\prod_{\gamma \in \Gamma^+} \prod_{g \in \mathrm{grd}(\gamma)} \Pr(E_g \mid E_J)\right) = \left(\prod_{\gamma \in \Gamma^+} \prod_{\substack{g \in \mathrm{grd}(\gamma) \\ J \models \neg g}} \exp(-W(\gamma))\right) \tag{11}$$

$$= \left(\prod_{\gamma \in \Gamma^+}\left(\prod_{\substack{g \in \mathrm{grd}(\gamma) \\ J \models \neg g}} \exp(-W(\gamma))\right) \times \left(\prod_{\substack{g \in \mathrm{grd}(\gamma) \\ J \models g}} \exp(-W(\gamma))\right) \times \left(\prod_{\substack{g \in \mathrm{grd}(\gamma) \\ J \models g}} \exp(W(\gamma))\right)\right)$$

$$= \left(\prod_{\gamma \in \Gamma^+}\left(\prod_{g \in \mathrm{grd}(\gamma)} \exp(-W(\gamma))\right) \times \left(\prod_{\substack{g \in \mathrm{grd}(\gamma) \\ J \models g}} \exp(W(\gamma))\right)\right)$$

$$= \left(\prod_{\gamma \in \Gamma^+} \prod_{g \in \mathrm{grd}(\gamma)} \exp(-W(\gamma))\right) \times \left(\prod_{\gamma \in \Gamma^+} \prod_{\substack{g \in \mathrm{grd}(\gamma) \\ J \models g}} \exp(W(\gamma))\right) \sim \exp\left(\sum_{\gamma \in \Gamma^+} \sum_{\substack{g \in \mathrm{grd}(\gamma) \\ J \models g}} W(\gamma)\right).$$

In the second equality listed previously, we simply multiply and divide by the same factor. Similarly, for soft rules with negative weights we have the following:

$$\left(\prod_{\gamma \in \Gamma^-} \prod_{g \in \mathrm{grd}(\gamma)} \Pr(E_{\neg g} \mid E_J)\right) = \left(\prod_{\gamma \in \Gamma^-}\left(\prod_{\substack{g \in \mathrm{grd}(\gamma) \\ J \models \neg(\neg g)}} \exp(-(-W(\gamma)))\right)\right)$$

$$= \left(\prod_{\gamma \in \Gamma^-} \prod_{\substack{g \in \mathrm{grd}(\gamma) \\ J \models g}} \exp(W(\gamma))\right). \tag{12}$$

Finally, we assume that J satisfies Ψ, and therefore, for $\gamma \in \Psi$ we have $\Pr(E_g \mid E_J) = 1$. Therefore, from Equations (11) and (12), we conclude that

$$\Pr(E_O \mid E_J) \sim \exp\left(\sum_{\gamma \in \Gamma} \sum_{\substack{g \in \mathrm{grd}(\gamma) \\ J \models g}} W(\gamma) \right).$$

As explained earlier in this section, this proportionality suffices to complete the proof.

6.2 Stochastic Context-Free Grammars in PPDL

We first recall the notion of a *Stochastic Context-Free Grammar* (*SCFG* for short), also known as *Probabilistic Context-Free Grammar* (*PCFG*). An SCFG is a tuple $(\mathbf{T}, \mathbf{N}, S, \rho)$, where:

— $\mathbf{T}$ and $\mathbf{N}$ are disjoint finite sets of *terminals* and *nonterminals*, respectively.
— $S \in \mathbf{N}$ is a *start symbol*.
— ρ is a function that maps every nonterminal $N \in \mathbf{N}$ to a finite probability distribution ρ_N over $(\mathbf{T} \cup \mathbf{N})^*$ (i.e., the set of all finite sequences of elements in $\mathbf{T} \cup \mathbf{N}$).

The semantics of an SCFG is a probability space over parse trees, where a *parse tree* is a (possibly infinite) directed and ordered tree such that the root is labeled with the start symbol S, each leaf is labeled with a terminal,[5] and each nonleaf is labeled with a nonterminal. The generative process that constructs a parse tree begins with the tree that consists of only S and repeatedly selects a leaf v (say, of a minimal distance from the root) with a nonterminal label N, randomly and independently selects a sequence $A_1, \ldots, A_m$ from the distribution ρ_N, and adds to v a sequence of m children with the labels $A_1, \ldots, A_m$ in order.

In this section, we show how to represent SCFGs in PPDL. Formally, we will define a PPDL program $\mathcal{P}$ so that every SCFG $(\mathbf{T}, \mathbf{N}, S, \rho)$ can be translated into an input I for $\mathcal{P}$ in such a way that the probability space over the parse trees is represented by $\mu_{\mathcal{P},I}$. For presentation's sake, we make two simplifying assumptions on the given SCFG. The first assumption is that $(\mathbf{T}, \mathbf{N}, S, \rho)$ is in epsilon-free *Chomsky Normal Form* (CNF); that is, for every nonterminal N and sequence $A_N^1 \ldots A_N^m$ in the support of ρ_N, either $m = 2$ and both A_N^1 and A_N^2 are nonterminals, or $m = 1$ and A_N^1 is a terminal (while $A_N^2 = \epsilon$ and ignored). The translation into general forms is straightforward but requires more technical details. The second assumption is that each ρ_N consists of *precisely* two sequences (which can be the same). Every SCFG can be changed to satisfy this requirement, by using extra nonterminals. Therefore, we will assume that ρ_N has the following form:

$$\rho_N(\alpha) = \begin{cases} p & \text{if } \alpha = A_N^1 A_N^2; \\ (1-p) & \text{if } \alpha = B_N^1 B_N^2. \end{cases}$$

In both cases, either both symbols are nonterminals or one of them is a terminal and the other the special symbol ϵ.

6.2.1 *The Program.*

We first need to explain how $(\mathbf{T}, \mathbf{N}, S, \rho)$ is represented in an EDB schema. Our EDB schema $\mathcal{E}$ has the following relation symbols:

— T and N are unary relation symbols that store terminals and nonterminals, respectively.
— Start is a unary relation that stores the start symbol S.

[5] For simplicity, we ignore the parse tree of the empty word.

- Rho represents ρ using tuples of the form $(N, A^1_N, A^2_N, B^1_N, B^2_N, p)$, stating that ρ_N selects $A^1_N A^2_N$ with probability p and $B^1_N B^2_N$ with probability $(1 - p)$.

From this, it is obvious how $(\mathbf{T}, \mathbf{N}, S, \rho)$ is represented in an input I over $\mathcal{E}$.

The IDB schema $\mathcal{I}$ represents a random parse tree. Each node v in the tree is identified by an integer i, with the start nonterminal having identifier 0. This integer is consistent with a left-to-right BFS traversal of the parse tree; that is, for two nodes v and v' with identifiers i and i', respectively, we have $i < i'$ if v' is deeper (i.e., farther from the root) than v, or if v and v' are on the same depth and v is left to v'. The IDB schema consists of two relations:

- V represents nodes v using tuples of the form (i, A), where i is the identifier of v and A is the (terminal or nonterminal) label of v.
- Choice represents the random choices associated with nonterminal nodes, using facts of the form (i, b), stating that the nonterminal node identified by i has selected $A^1_N A^2_N$ (if $b = 0$) or $B^1_N B^2_N$ (if $b = 1$).

The PPDL program $\mathcal{P}$ is the following:

(1)	$\mathrm{V}(0, A)$	$\leftarrow$	$\mathrm{Start}(A)$
(2)	$\mathrm{Choice}(i, \mathsf{Flip}[p])$	$\leftarrow$	$\mathrm{V}(i, A), \mathrm{N}(A), \mathrm{Rho}(A, \_, \_, \_, p)$
(3)	$\mathrm{V}(2i + 1, A^1)$	$\leftarrow$	$\mathrm{V}(i, A), \mathrm{Choice}(i, 0), \mathrm{Rho}(A, A^1, \_, \_, \_)$
(4)	$\mathrm{V}(2i + 2, A^2)$	$\leftarrow$	$\mathrm{V}(i, A), \mathrm{Choice}(i, 0), \mathrm{Rho}(A, A^1, A^2, \_, \_), \mathrm{N}(A^1)$
(5)	$\mathrm{V}(2i + 1, B^1)$	$\leftarrow$	$\mathrm{V}(i, A), \mathrm{Choice}(i, 1), \mathrm{Rho}(A, \_, \_, B^1, \_)$
(6)	$\mathrm{V}(2i + 2, B^2)$	$\leftarrow$	$\mathrm{V}(i, A), \mathrm{Choice}(i, 1), \mathrm{Rho}(A, \_, \_, B^1, B^2), \mathrm{N}(B^1)$

For readability's sake, we use the convention of replacing a variable that occurs only once with underscore (_). The first rule constructs the root, which corresponds to the start symbol. The second rule states that every node i with a nonterminal label A needs to make a random choice of a sequence from the distribution ρ_A. The third rule says that if a node i with the nonterminal label A has chosen the left sequence $A^1 A^2$, then we construct a child u of i with the label A^1; the identifier of u is $2i + 1$. The fourth rule does the same, except that u has a greater index $(2i + 2)$ in the identifier, and it is constructed only if A^1 is a nonterminal. The choice of the indices $2i + 1$ and $2i + 2$ guarantees the correspondence to BFS scanning. The fifth and sixth rules are the same as the third and fourth, respectively, except that they apply to the case where node i has chosen the right sequence $B^1 B^2$.

Our Datalog model does not allow arithmetic operations such as $2i + 1$ and $2i + 2$. Nevertheless, these can be simulated as special cases of (deterministic) numerical distributions.

6.2.2 Accounting for Input Strings. In practice, an SCFG is used for representing the probability space of parses of a given input string (Klein and Manning 2003). Formally, for a given string $\mathbf{s}$ of terminals, the probability space is conditioned on two properties:

(1) The parse tree is finite.
(2) The sequence of leaf terminals, from left to right, forms the string $\mathbf{s}$.

Next, we will show how to realize these conditions in our PPDL program $\mathcal{P}$.

We represent the string $\mathbf{s}$ using a binary EDB Str that stores tuples of the form (j, c), stating that the character at position j is c. For a technical reason, we also need the unary EDB Len that stores the length of Str (in a single tuple). For example, the string PPDL will be represented using the facts Str(1, P), Str(2, P), Str(3, D), Str(4, L), and Len(4). The following set of rules constructs the IDB relation $\mathrm{SameStr}(i, j, k)$, stating that the subtree under the nonterminal node i spans the substring from position j to position k:

$$\begin{aligned}\mathrm{SameStr}(i, j, j) &\leftarrow \mathrm{V}(i, \_), \mathrm{V}(2i + 1, c), \mathrm{Str}(j, c)\\ \mathrm{SameStr}(i, j, k) &\leftarrow \mathrm{SameStr}(2i + 1, j, \ell), \mathrm{SameStr}(2i + 2, \ell + 1, k).\end{aligned}$$

The first rule says that a nonterminal node i spans the terminal at position j if i is the parent of the only (terminal) node with the label c. The second rule states that i spans the substring from j to k if i has two children, which must be $2i + 1$ and $2i + 2$, such that for some ℓ it holds that the first child spans the substring from j to ℓ and the second spans the substring from $\ell + 1$ to k. We then add the following condition, stating that the root spans the entire string:

$$\mathrm{Len}(n) \rightarrow \mathrm{SameStr}(0, 1, n).$$

Recall that 0 is the identifier of the root. It can be shown (e.g., by induction on $k - j$) that if we have an entry $\mathrm{SameStr}(i, j, k)$, then the subtree underneath node i is finite, and moreover, it indeed spans the substring from j to k. Hence, the previous condition captures precisely the condition of parsing the input string.

7 RELATED WORK

In this section, we compare PPDL to related formalisms from the literature and contrast them with our work. We consider several categories of formalisms. For the first three categories, we adopt the classification of De Raedt and Kimmig 2015. These formalisms fall in the broader category of *Statistical Relational Learning* (SRL). For a thorough survey on SRL, we refer the reader to De Raedt et al. 2016.

7.1 Imperative Specifications

The first category includes imperative probabilistic programming languages (Roy 2014). We also include in this category declarative specifications of Bayesian networks, such as *Context-Sensitive Probabilistic Knowledge* (Ngo and Haddawy 1997) and BLOG (Milch et al. 2005). Although declarative in nature, these languages inherently assume a form of acyclicity that allows the rules to be executed in a serial manner. 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 4, for example, uses recursion (as is typically the case for probabilistic models in social network analysis), and so is the translation from SCFGs that we discussed in Section 6.2. In particular, it is not clear how the virus program can be phrased by translation into a Bayesian network, and SCFG (that may define an uncountable probability space (Etessami and Yannakakis 2009)) cannot be translated in Bayesian networks. 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. Related models are those of MCDB (Jampani et al. 2008) and SimSQL (Cai et al. 2013) that propose SQL extensions coupled with Monte Carlo simulations and parallel database techniques for stochastic analytics in the database. Again, these formalisms assume an order over the execution of rules, hence their association with the first category.

7.2 Logic over Probabilistic Databases

The formalisms in the second category, also referred to as the *distributional semantics* (Raedt and Kimmig 2015), 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 (Kimmig et al. 2011) or Datalog (Abiteboul et al. 2014), is evaluated over the resulting random structure. This approach has been taken by PRISM (Sato and Kameya 1997), and to a large extent by *probabilistic databases* (Suciu et al. 2011) and their semistructured counterparts (Kimelfeld and Senellart 2013). The Dyna (Eisner and Filardo 2010) system has adopted this approach to extend the semantics of Datalog with randomness. The focus of our work is 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 (Widom 2008), I-SQL, and world-set algebra (Antova et al. 2007a, 2007b). The latter two are natural analogs to SQL and relational algebra for the case of incomplete information and probabilistic data (Antova et al. 2007a). 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 subqueries. World-set algebra has been extended to (world-set) Datalog, fixpoint, and while-languages (Deutch et al. 2010) to define Markov chains. While such languages cannot explicitly specify probability distributions, they may simulate a specific categorical distribution indirectly using nontrivial programs with specialized language constructs like `repair-key` on input tuples with weights representing samples from the distribution.

7.3 Model Representation via Formula Grounding

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 MLNs (Domingos and Lowd 2009; Niu et al. 2011) (which we discussed in detail in Section 6.1), 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 (Niu et al. 2012), 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 concept similar to MLN is defined by the *hybrid relational dependency networks* (Ravkic et al. 2015), where each grounding of a rule defines a factor as a probability distribution over a random variable that might be continuous. In *Probabilistic Soft Logic* (Bröcheler et al. 2010), in each possible world every fact is associated with a degree of truth.

Further formalisms in this category are *probabilistic Datalog* (Fuhr 2000) and *probabilistic Datalog+/-* (Gottlob et al. 2013). There, every rule is associated with a probability. The semantics of the former is different from ours and that of the other formalisms mentioned thus far. There, 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.[6] Besides our support for

[6]Another variant of Probabilistic Datalog+/- (Riguzzi et al. 2012) adopts the distributional semantics and belongs in the former category (Logic over Probabilistic Databases).

numerical probability distributions, our formalism is used for defining a single probability space, which is on par with the standard practice in probabilistic programming.

7.4 Recursion Models

As discussed earlier, GDatalog[Δ] allows for recursion, and the semantics is captured by (possibly infinite) Markov chains. Related formalisms are that of SCFGs and the more general *Recursive Markov Chains* (RMCs) (Etessami and Yannakakis 2009), 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, SCFGs and RMCs are used in the context of probabilistic XML (Cohen and Kimelfeld 2010; Benedikt et al. 2010). These formalisms do not involve numerical distributions. We have shown a translation from SCFGs to PPDL. In future work, we plan to further study the relative expressive power of these models, compared to restrictions of our framework.

7.5 Probabilistic Datalog Models

We now discuss the formalisms that we view as closest to ours. Like PPDL, these formalisms propose Datalog with deterministic bodies and probabilistic conclusions such that the random choice is applied upon each firing of a rule, and they make no assumption of acyclicity or stratifiability on the programs. These include the *Constraint Logic Programming*, denoted *CLP(BN)* (Costa et al. 2003); the *Independent Choice Logic* (*ICL*) (Poole 2008); *P-log* (Baral et al. 2009); the *Causal Probabilistic logic* (*CP-logic*) (Vennekens et al. 2009); and the *Distributional Clauses* (*DC*) (Gutmann et al. 2011; Nitti et al. 2016).

Yet, each of these models makes substantial assumptions in order to define the probabilistic semantics, assumptions that we are able to avoid in PPDL. In CLP(BN), the assumption is that each random value can be tracked to precisely one grounded rule that generates it. In particular, the semantics is not defined when a random value can be generated by two different groundings of a rule (when the distribution may entail different parameters each time), as in the case where existential variables appear in the body. Moreover, it is not clear how their semantics can be extended to guarantee independence of the chase order.

In ICL, the assumption is that the rules are acyclic in their *grounded* version. That is, there is assignment of a natural number (stratum) to each ground atom so that the number assigned to the head of each clause is greater than those of the atoms in the body. For instance, the natural program that tests whether a graph contains a cycle is *not* acyclic in this sense. In DC, this assumption is weakened to the requirement of being "distribution stratified." This means that the number assigned to the head of each clause is *greater than or equal* to those of the atoms in the body (hence, recursion is allowed within a stratum), but distributional heads should have a strictly higher number. Yet, it is not clear how stratification at the ground level can be verified. As an example, consider the following PPDL program, simulating announcement propagation in a social network:

$$\begin{aligned}\mathrm{Shares}(x, \mathsf{Flip}[0.2]) &\leftarrow \mathrm{Knows}(x)\\ \mathrm{Knows}(x) &\leftarrow \mathrm{Follows}(x,y), \mathrm{Shares}(y,1).\end{aligned}$$

The program states that if a person knows about the announcement, then he or she shares it with probability 0.2, and he or she knows about the announcement whenever a friend of his or hers shares it. In DC, the corresponding program is the following:

$$\begin{aligned}\mathrm{Shares}(x) \sim [0.2:1, 0.8:0] &\text{ :- } \mathrm{Knows}(x)\\ \mathrm{Knows}(x) &\text{ :- } \mathrm{Follows}(x,y), \simeq(\mathrm{Shares}(y)) \text{ is } 1.\end{aligned}$$

In this program, Shares(z) represents a random variable (for every z), and $\simeq$(Shares(z)) stands for its sampled value. Now, suppose that we have two people that follow each other, say, Alice and Bob. The stratum assigned to the head Shares(Alice) $\sim [0.2 : 1, 0.8 : 0]$ should be higher than that of Knows(Alice), which should be at least that of $\simeq$(Shares(Bob)). Moreover, DC requires the stratum of $\simeq$(Shares(Bob)) to be at least that of Shares(Bob) $\sim [0.2 : 1, 0.8 : 0]$. Hence, the stratum of Shares(Alice) $\sim [0.2 : 1, 0.8 : 0]$ should be higher than that of Shares(Bob) $\sim [0.2 : 1, 0.8 : 0]$. Yet, due to symmetry, the same should hold when replacing Alice and Bob. Therefore, the aforementioned DC program is, in fact, illegal. We mention here that DC supports significant features that PPDL does not, such as continuous numerical distributions and negation.

In P-log, several assumptions are made (phrased as *Conditions 1–3*) that boil down to saying that in every possible world a random value can be mapped to precisely one ground rule that produced it, but it is not clear how such conditions can be verified. In CP-logic, the assumption is that each ground rule can fire at most once. In particular, by adding to a program a rule that is equivalent (or even identical) to an existing rule, we may change the semantics of the program, since the new rule may fire additional random choices. Hence, CP-logic is not invariant under equivalence-preserving transformations.

Another significant difference from P-log and CP-logic is that in each of the two, the number of random choices is determined by the number of groundings, and in fact, such programs can be translated into ProbLog. In particular, the translation from SCFG that we showed in Section 6.2 can be carried out in neither P-log nor CP-logic.

8 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 work 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* (Raiffa and Schlaifer 1961); 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 (Antova et al. 2007a, 2007b). 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 such popular distributions are multinomial, Dirichlet, and multivariate Gaussian distribution.

We are working on extending LogiQL (Halpin and Rugaber 2014), the programming language of LogicBlox (Aref et al. 2015), 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. Toward 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 and factor-graph libraries). 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.

REFERENCES

Serge Abiteboul, Daniel Deutch, and Victor Vianu. 2014. Deduction with contradictions in datalog. In *International Conference on Database Theory*. 143–154.

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

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

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

Lyublena Antova, Christoph Koch, and Dan Olteanu. 2007b. Query language support for incomplete information in the MayBMS system. In *Very Large Data Bases*. 1422–1425.

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

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

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

Vince Bárány, Balder ten Cate, Benny Kimelfeld, Dan Olteanu, and Zografoula Vagena. 2016. Declarative probabilistic programming with datalog. In *International Conference on Database Theory*, Vol. 48. 7:1–7:19.

Michael Benedikt, Evgeny Kharlamov, Dan Olteanu, and Pierre Senellart. 2010. Probabilistic XML via markov chains. In *Proceedings of the VLDB Endowment* 3, 1 (2010), 770–781.

David M. Blei, Andrew Y. Ng, and Michael I. Jordan. 2003. Latent Dirichlet allocation. *J. Machine Learn. Res.* 3 (2003), 993–1022.

Matthias Bröcheler, Lilyana Mihalkova, and Lise Getoor. 2010. Probabilistic similarity logic. In *Uncertainty in Artificial Intelligence*. 73–82.

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

Bob Carpenter, Andrew Gelman, Matthew Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Michael A. Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2017. Stan: A probabilistic programming language. *J. Stat. Softw.* 76, 1 (2017), 1–32.

Sara Cohen and Benny Kimelfeld. 2010. Querying parse trees of stochastic context-free grammars. In *International Conference on Database Theory*. ACM, 62–75.

Vítor Santos Costa, David Page, Maleeha Qazi, and James Cussens. 2003. CLP(BN): Constraint logic programming for probabilistic knowledge. In *Uncertainty in Artificial Intelligence*. 517–524.

Mary Kathryn Cowles. 2013. *Applied Bayesian Statistics: With R and OpenBUGS Examples*. Vol. 98. Springer Science & Business Media.

Daniel Deutch, Christoph Koch, and Tova Milo. 2010. On probabilistic fixpoint and Markov chain query languages. In *Symposium on Principles of Database Systems*. 215–226.

Pedro Domingos and Daniel Lowd. 2009. *Markov Logic: An Interface Layer for Artificial Intelligence*. Morgan & Claypool Publishers.

Jason Eisner and Nathaniel Wesley Filardo. 2010. Dyna: Extending datalog for modern AI. In *Datalog Reloaded - 1st International Workshop (Datalog'10). Revised Selected Papers (Lecture Notes in Computer Science)*, Vol. 6702. Springer, 181–220.

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

Kousha Etessami and Mihalis Yannakakis. 2009. Recursive markov chains, stochastic grammars, and monotone systems of nonlinear equations. *J. ACM* 56, 1 (2009), 1:1–1:66.

Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. 2005. Data exchange: Semantics and query answering. *Theor. Comput. Sci.* 336, 1 (2005), 89–124. DOI:http://dx.doi.org/10.1016/j.tcs.2004.10.033

Nissim Francez. 1986. *Fairness*. Springer. DOI:http://dx.doi.org/10.1007/978-1-4612-4886-6

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

Noah D. Goodman. 2013. The principles and practice of probabilistic programming. In *Symposium on Principles of Programming Languages*. 399–402.

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

Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. 1993. Maintaining views incrementally. In *SIGMOD*. 157–166.

Bernd Gutmann, Ingo Thon, Angelika Kimmig, Maurice Bruynooghe, and Luc De Raedt. 2011. The magic of logical inference in probabilistic programming. *TPLP* 11, 4–5 (2011), 663–680. DOI:http://dx.doi.org/10.1017/S1471068411000238

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

Shawn Hershey, Jeffrey Bernstein, Bill Bradley, Andrew Schweitzer, Noah Stein, Theophane Weber, and Benjamin Vigoda. 2012. Accelerating inference: Towards a full language, compiler and hardware stack. *CoRR* abs/1212.2991 (2012).

Daniel Roy. 2014. Repository on probabilistic programming languages. (2014). http://www.probabilistic-programming.org.

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

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

Abhay Kumar Jha and Dan Suciu. 2012. Probabilistic databases with markoviews. *Proceedings of the VLDB Endowment* 5, 11 (2012), 1160–1171.

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

Angelika Kimmig, Bart Demoen, Luc De Raedt, Vitor Santos Costa, and Ricardo Rocha. 2011. On the implementation of the probabilistic logic programming language ProbLog. *Theory Pract. Log. Program.* 11 (2011), 235–262. DOI:http://dx.doi.org/10.1017/S1471068410000566

Dan Klein and Christopher D. Manning. 2003. Accurate unlexicalized parsing. In *Meeting of the Association for Computational Linguistics*. 423–430.

John W. Lloyd. 1987. *Foundations of Logic Programming*. 2nd ed. Springer.

Jorge Lobo, Jack Minker, and Arcot Rajasekar. 1992. *Foundations of Disjunctive Logic Programming*. MIT Press.

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

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

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

Andrew McCallum. 1999. Multi-label text classification with a mixture model trained by EM. In *Association for the Advancement of Artificial Intelligence Workshop on Text Learning*.

Brian Milch, Bhaskara Marthi, Stuart J. Russell, David Sontag, Daniel L. Ong, and Andrey Kolobov. 2005. BLOG: Probabilistic models with unknown objects. In *International Joint Conference on Artificial Intelligence*. Professional Book Center, 1352–1359.

Boris Motik, Yavor Nenov, Robert Edgar Felix Piro, and Ian Horrocks. 2015. Incremental update of datalog materialisation: The backward/forward algorithm. In *Association for the Advancement of Artificial Intelligence*. 1560–1568.

Liem Ngo and Peter Haddawy. 1997. Answering queries from context-sensitive probabilistic knowledge bases. *Theor. Comput. Sci.* 171, 1–2 (1997), 147–177. DOI: http://dx.doi.org/10.1016/S0304-3975(96)00128-4

Kamal Nigam, Andrew McCallum, Sebastian Thrun, and Tom M. Mitchell. 2000. Text classification from labeled and unlabeled documents using EM.*Machine Learning* 39, 2–3 (2000), 103–134.

Davide Nitti, Tinne De Laet, and Luc De Raedt. 2016. Probabilistic logic programming for hybrid relational domains. *Mach. Learn.* 103, 3 (2016), 407–449. DOI: http://dx.doi.org/10.1007/s10994-016-5558-8

Feng Niu, Christopher Ré, AnHai Doan, and Jude W. Shavlik. 2011. Tuffy: Scaling up statistical inference in markov logic networks using an RDBMS. In *Proceedings of the VLDB Endowment* 4, 6 (2011), 373–384.

Feng Niu, Ce Zhang, Christopher Ré, and Jude W. Shavlik. 2012. DeepDive: Web-scale knowledge-base construction using statistical learning and inference. In *International Workshop on Searching and Integrating New Web Data Sources (CEUR Workshop Proc.)*, Vol. 884. 25–28.

Aditya V. Nori, Chung-Kil Hur, Sriram K. Rajamani, and Selva Samuel. 2014. R2: An efficient MCMC sampler for probabilistic programs. In *Association for the Advancement of Artificial Intelligence*. 2476–2482.

Brooks Paige and Frank Wood. 2014. A compilation target for probabilistic programming languages. In *International Conference on Machine Learning*, Vol. 32. 1935–1943.

Anand Patil, David Huard, and Christopher J. Fonnesbeck. 2010. PyMC: Bayesian stochastic modelling in python. *J. Stat. Softw.* 35, 4 (2010), 1–81.

Judea Pearl. 1989. *Probabilistic Reasoning in Intelligent Systems - Networks of Plausible Inference*. Morgan Kaufmann.

Avi Pfeffer. 2009. *Figaro: An Object-oriented Probabilistic Programming Language*. Technical Report. Charles River Analytics. 137 pages.

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

Luc De Raedt, Kristian Kersting, Sriraam Natarajan, and David Poole. 2016. *Statistical Relational Artificial Intelligence: Logic, Probability, and Computation*. Morgan & Claypool Publishers. DOI: http://dx.doi.org/10.2200/S00692ED1V01Y201601AIM032

Luc De Raedt and Angelika Kimmig. 2015. Probabilistic (logic) programming concepts. *Mach. Learn.* 100, 1 (2015), 5–47.

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

Irma Ravkic, Jan Ramon, and Jesse Davis. 2015. Learning relational dependency networks in hybrid domains. *Mach. Learn.* 100, 2–3 (2015), 217–254. DOI: http://dx.doi.org/10.1007/s10994-015-5483-2

Fabrizio Riguzzi, Elena Bellodi, and Evelina Lamma. 2012. Probabilistic ontologies in datalog+/-. In *Italian Conference on Computational Logic*. 221–235.

Taisuke Sato and Yoshitaka Kameya. 1997. PRISM: A language for symbolic-statistical modeling. In *International Joint Conference on Artificial Intelligence*. 1330–1339.

Oded Shmueli. 1993. Equivalence of datalog queries is undecidable. *J. Logic Progr.* 15, 3 (1993), 231–241.

Parag Singla and Pedro M. Domingos. 2006. Entity resolution with Markov logic. In *IEEE International Conference on Data Mining*. 572–582.

Dan Suciu, Dan Olteanu, Christopher Ré, and Christoph Koch. 2011. *Probabilistic Databases*. Morgan & Claypool Publishers.

Nachum Dershowitz. 2005. *Term Rewriting Systems*. Marc Bezem, Jan Willem Klop, and Roel De Vrijer (Eds.). Cambridge University Press, Cambridge Tracts in Theoretical Computer Science 55, Vol.5. 395–399.

Joost Vennekens, Marc Denecker, and Maurice Bruynooghe. 2009. CP-logic: A language of causal probabilistic events and its relation to logic programming. *TPLP* 9, 3 (2009), 245–308. DOI: http://dx.doi.org/10.1017/S1471068409003767

Jennifer Widom. 2008. Trio: A system for data, uncertainty, and lineage. In *Managing and Mining Uncertain Data*, Charu Aggarwal (Ed.). Springer-Verlag, Chapter 5.

Received August 2016; revised May 2017; accepted August 2017