Declarative Probabilistic Logic Programming in Discrete-Continuous Domains

Pedro Zuidberg Dos Martires[a], Luc De Raedt[a,b,c], Angelika Kimmig[b,c]

[a]*Centre for Applied Autonomous Sensor Systems, Örebro University, Sweden*
[b]*Department of Computer Science, KU Leuven, Belgium*
[c]*Leuven.AI, Belgium*

Abstract

Over the past three decades, the logic programming paradigm has been successfully expanded to support probabilistic modeling, inference and learning. The resulting paradigm of probabilistic logic programming (PLP) and its programming languages owes much of its success to a declarative semantics, the so-called distribution semantics. However, the distribution semantics is limited to discrete random variables only. While PLP has been extended in various ways for supporting hybrid, that is, mixed discrete and continuous random variables, we are still lacking a declarative semantics for hybrid PLP that not only generalizes the distribution semantics and the modeling language but also the standard inference algorithm that is based on knowledge compilation. We contribute the hybrid distribution semantics together with the hybrid PLP language DC-ProbLog and its inference engine *infinitesimal algebraic likelihood weighting* (IALW). These have the original distribution semantics, standard PLP languages such as ProbLog, and standard inference engines for PLP based on knowledge compilation as special cases. Thus, we generalize the state-of-the-art of PLP towards hybrid PLP in three different aspects: semantics, language and inference. Furthermore, IALW is the first inference algorithm for hybrid probabilistic programming based on knowledge compilation.

Keywords: Probabilistic Programming, Declarative Semantics, Discrete-Continuous Distributions, Likelihood Weighting, Logic Programming, Knowledge Compilation, Algebraic Model Counting

1. Introduction

Probabilistic logic programming (PLP) is at the crossroads of two parallel developments in artificial intelligence and machine learning. On the one hand, there are the probabilistic programming languages with built-in support for machine learning. These languages can be used represent very expressive – Turing equivalent – probabilistic models, and they provide primitives for inference and learning. On the other hand, there is the longstanding open question for integrating the two main frameworks for reasoning, that is logic and probability, within a common framework [Russell, 2015, De Raedt et al., 2016]. Probabilistic logic programming [De Raedt and Kimmig, 2015, Riguzzi, 2018] fits both paradigms and goes back to at least the early 90s with seminal works by Sato [1995] and Poole [1993]. Poole introduced ICL, the Independent Choice Logic, an elegant extension of the Prolog programming language, and Sato introduced the *distribution semantics* for probabilistic logic programs in conjunction with a learning algorithm based on expectation maximization (EM). The PRISM language [Sato, 1995], which utilizes the distributions semantics and the EM learning algorithm constitutes, to the best of the authors' knowledge, the very first probabilistic programming language with support for machine learning.

Today, there is a plethora of probabilistic logic programming languages, most of which are based on extensions of the ideas by Sato and Poole [Sato and Kameya, 1997, Kersting and De Raedt, 2000, Vennekens et al., 2004, De Raedt et al., 2007]. However, the vast majority of them is restricted to discrete, and more precisely finite categorical, random variables. When merging logic with probability, the restriction to discrete random variables is natural and allowed Sato to elegantly extend the logic program semantics into the celebrated distribution semantics. However, it is also an important restriction, which raises the question of how to extend the semantics towards hybrid, i.e. discrete-continuous, random variables.

Defining the semantics of probabilistic programming language with support for random variables with infinite and possibly uncountable sample spaces is a much harder task. This can be observed when looking at the development of important imperative and functional probabilistic programming languages [Goodman et al., 2008, Mansinghka et al., 2014] that support continuous random variables. These works initially focused on inference, typically using a particular Monte Carlo approach, yielding an operational or procedural semantics. It is only follow-up work that started to address a declarative semantics for such hybrid probabilistic programming languages. [Staton et al., 2016, Wu et al., 2018].

The PLP landscape has experienced similar struggles. First approaches for hybrid PLP languages were achieved by restricting the language [Gutmann et al., 2010, 2011, Islam et al., 2012] or via recourse to procedural semantics [Nitti et al., 2016]. The key contributions of this paper are:

C1 We introduce the *hybrid* distribution semantics for mixed discrete-continuous probabilistic logic programming. The *hybrid* distribution semantics extends Sato's distribution semantics and supports:

- a countably infinite number of random variables,
- a uniform treatment of discrete and continuous random variables,
- a clear separation between probabilistic dependencies and logical dependencies by extending the ideas of Poole [2010] to the hybrid domain.

C2 We introduce DC-ProbLog, an expressive PLP language in the discrete-continuous domain, which incorporates the *hybrid* distribution semantics. DC-ProbLog has standard discrete PLP, e.g. ProbLog [Fierens et al., 2015], as a special case (unlike other hybrid PLP languages [Gutmann et al., 2011, Nitti et al., 2016]).

C3 We introduce a novel inference algorithm, *infinitesimal algebraic likelihood weighting* (IALW), for hybrid PLPs, which extends the standard knowledge compilation approach used in PLP towards mixed discrete continuous distributions, and which provides an operational semantics for hybrid PLP.

In essence, our contributions **C1** and **C2** generalize both Sato's distribution semantics and discrete PLP such that in the absence of random variables with infinite sample spaces we recover the ProbLog language and declarative semantics. It is noteworthy that our approach of disentangling probabilistic dependencies and logical ones, allows us to express more general distributions than state-of-the-art approaches such e.g. [Gutmann et al., 2011, Nitti et al., 2016, Azzolini et al., 2021]. Contribution **C3** takes this generalization to the inference level: in the exclusive presence of finite random variables our IALW algorithm reduces to ProbLog's current inference algorithm [Fierens et al., 2015].

2. A Panoramic Overview

Before diving into the technical details of the paper we first give a high-level overview of the DC-ProbLog language. This will also serve us as roadmap to the remainder of the paper. We will first introduce, by example, the DC-ProbLog language (Section 2.1). The formal syntax and semantics of which are discussed in Section 3 and Section 4. In Section 2.2 we demonstrate how to perform probabilistic inference in DC-ProbLog by translating a queried DC-ProbLog program to an algebraic circuit [Zuidberg Dos Martires et al., 2019a]. Before giving the details of this transformation in Section 6 and Section 7, we define conditional probability queries on DC-ProbLog programs (Section 5). The paper ends with a discussion on related work (Section 8) and concluding remarks in Section 9.

Throughout the paper, we assume that the reader is familiar with basic concepts from logic programming and probability theory. We provide, however, a brief refresher of basic logic programming concepts in Appendix A. In Appendix B we give a tabular overview of notations used, and in the remaining sections of the appendix we give proofs to propositions and theorems or discuss is in more detail some of the more subtle technical details.

2.1. Panorama of the Syntax and Semantics

Example 2.1. *A shop owner creates random bags of sweets with two independent random binary properties (*large *and* balanced*). He first picks the number of red sweets from a Poisson distribution whose parameter is 20 if the bag is large and 10 otherwise, and then the number of yellow sweets from a Poisson whose parameter is the number of red sweets if the bag is balanced and twice that number otherwise. His favorite type of bag contains more than 15 red sweets and no less than 5 yellow ones. We model this in DC-ProbLog as follows:*

```
0.5::large.
0.5::balanced.

```

```
4  red ~ poisson(20) :- large.
5  red ~ poisson(10) :- not large.
6
7  yellow ~ poisson(red) :- balanced.
8  yellow ~ poisson(2*red) :- not balanced.
9
10 favorite :- red > 15, not yellow < 5.
```

In the first two lines we encounter probabilistic facts*, a well-known modelling construct in discrete PLP languages (e.g. [De Raedt et al., 2007]). Probabilistic facts, written as logical facts labeled with a probability, express Boolean random variables that are true with the probability specified by the label. For instance,* `0.5::large` *expresses that* `large` *is true with probability* `0.5` *and false with probability* `1-0.5`*.*

In Lines 4 to 8, we use distributional clauses *(DCs); introduced by Gutmann et al. [2011] into the PLP literature. DCs are of the syntactical form* `v~d:-b` *and define random variables* `v` *that are distributed according to the distribution* `d`*, given that* `b` *is true. For example, Line 4 specifies that when* `large` *is true,* `red` *is distributed according to a Poisson distribution. We call the left-hand argument of a* ~/2 *predicate in infix notation a* random term*. The random terms in the program above are* `red` *and* `yellow`*.*

Note how random terms reappear in three distinct places in the DC-ProbLog program. First, we can use them as parameters to other distributions, e.g. `yellow ~ poisson(red)`*. Second, we can use them within arithmetic expression, such as* `2*red` *in Line 8. Third, we can use them in comparison atoms (*`red>15`*) in Line 10. The comparison atoms appear in the bodies of logical rules that express logical consequences of probabilistic event, for example having more than 15 red sweets and less than 5 yellow ones.*

Probabilistic facts and distributional clauses are the main modelling constructs to define random variables in probabilistic logic programs. As they are considered to be fundamental building blocks of a PLP language, the semantics of a language are defined in function of these syntactical constructs (cf.. [Fierens et al., 2015, Gutmann et al., 2011]). We now make an important observation: probabilistic facts and distributional clauses can be deconstructed into a much more fundamental concept, which we call the *distributional fact*. Syntactically, a distributional fact is of the form `v~d`. That is, a distributional clause with an empty body. As a consequence, probabilistic facts and distributional clauses do not constitute fundamental concepts in PLP but are merely special cases, i.e. while helpful for writing concise programs, they are only of secondary importance when it comes to semantics.

Example 2.2. *We now rewrite the program in Example 2.1 using distributional facts only. Note how probabilistic facts are actually distributional facts in disguise. The random variable is now distributed according to a Bernoulli distribution (*`flip`*) and the atom of the probabilistic fact is the head of a rule with a probabilistic comparison in its body (e.g. Lines 1 and 2 in the program below). Rewriting distributional facts is more involved. The main idea is to introduce a distinct random term for each distributional clause. Take for example, the random term* `red` *in Example 2.1. This random* *term encodes, in fact, two distinct random variables, which we denote in the program below* `red_large` *and* `red_small`. *We now have to propagate this rewrite through the program and replace every occurrence of* `red` *with* `red_large` *and* `red_small`. *This is why we get instead of two distributional clauses for* `yellow`*, four distributional facts. It explains also why we get instead of one rule for* `favorite` *in Example 2.1 four rules now.*

```
rv_large ~ flip(0.5).
large :- rv_large=:=1.
rv_balanced ~ flip(0.5).
balanced :- rv_balanced=:=1.

red_large ~ poisson(20).
red_small ~ poisson(10).

yellow_large_balanced ~ poisson(red_large).
yellow_large_unbalanced ~ poisson(2*red_large).
yellow_small_balanced ~ poisson(red_small).
yellow_small_unbalanced ~ poisson(2*red_small).

favorite :- large, red_large > 15,
              balanced, not yellow_large_balanced < 5.
favorite :- large, red_large > 15,
              not balanced, not yellow_large_unbalanced < 5.
favorite :- not large, red_small > 15,
              balanced, not yellow_small_balanced < 5.
favorite :- not large, red_small > 15,
              not balanced, not yellow_small_unbalanced < 5.
```

The advantage of using probabilistic facts and distributional clauses is clear. They allow us to write much more compact and readable programs. However, as they do not really constitute fundamental building blocks of PLP, defining the semantics of a PLP language is much more intricate. For this reason we adapt a two-stage approach to define the semantics of DC-ProbLog. We first define the semantics of DF-PLP, a bare-bones language with no syntactic sugar only relying on distributional facts to define random variables. This happens in Section 3. During the second stage we define the program transformations to rewrite syntactic sugar (e.g. distributional clauses) as distributional facts. The semantics of DF-PLP and the program transformations then give us the semantics of the DC-ProbLog language (cf. Section 4).

2.2. Panorama of the Inference

The part of the paper concerning inference consists of three sections. First, we start in Section 5 to define a query to a DC-ProbLog program. For instance, we might be interested in the probability

$$P(\texttt{favorite} = \top, \texttt{large} = \bot)$$

In other words, the joint probability of favorite being true and large being false. While the example query above is simply a joint probability, we generalize this in Section 5 to conditional probabilities (possible with zero-probability events in the conditioning set).

Second, we map the queried ground program to a labeled Boolean formula. Contrary to the approach taken by Fierens et al. [2015] the labels are not probabilities (as usual in PLP) but indicator functions. This mapping to a labeled Boolean formula happens again in a series of program transformations, which we describe in Section 6. On of these steps is obtaining the relevant ground program to a query. For example, for the query above only the last two rules for favorite matter.

```
favorite :- not large, rs > 15, balanced, not ysb < 5.
favorite :- not large, rs > 15, not balanced, not ysu < 5.
```

Here, we abbreviated red_small as rs and yellow_small_balanced and yellow_small_unbalanced as ysb and ysu, respectively. We can further rewrite these rules by replacing large and balanced with equivalent comparison atoms and pushing the negation into the comparisons:

```
favorite :- rvl=:=0, rs > 15, rvb=:=1, ysb >= 5.
favorite :- rvl=:=0, rs > 15, rvb=:=0, ysu >= 5.
```

Again using abbreviations: rvl for rv_large and rvb for rv_balanced.

In Section 7 we then show how to compute the expected value of the labeled propositional Boolean formula corresponding to these rules by compiling it into an algebraic circuit, which is graphically depicted in Figure 2.1. In order to evaluate this circuit and obtain the queried probability (expected value), we introduce the IALW algorithm.

The idea of IALW is the following: sample the random variables dangling at the bottom of the circuit by sampling parents before children. For instance, we first sample from poisson(10) (at the very bottom) before sampling from poisson(rs) using the sampled value of the parent as the parameter of the child. Once we reach the comparison atoms (e.g. $ysb \geq 5$) we stick in the sampled values for the mentioned random variables. This evaluates the comparisons to either 1 or 0, for which we then perform the sums and products as prescribed by the circuit. We get a Monte Carlo estimate of the queried probability by averaging over multiple such evaluations of the circuit.

The method, as sketched here, is in essence the probabilistic inference algorithm Sampo presented in [Zuidberg Dos Martires et al., 2019b]. The key contribution of IALW, which we discuss in Section 7, is to extend Sampo such that conditional inference with zero probability events is performed correctly.

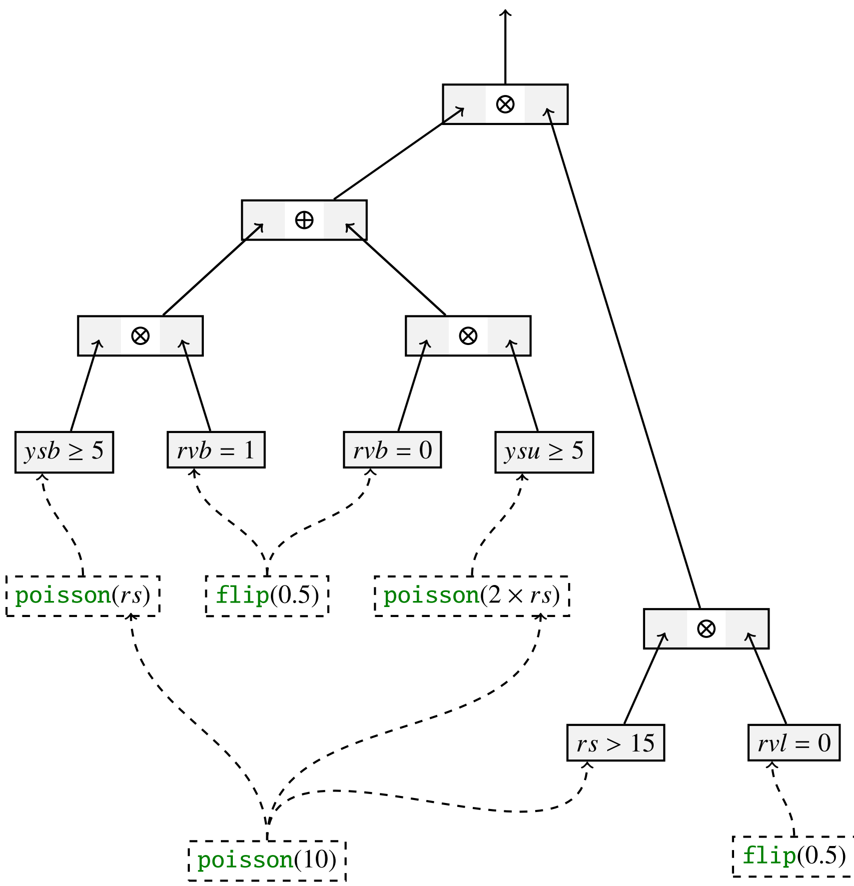

Figure 2.1: Graphical representation of the computation graph (i.e. algebraic circuit) used to compute the probability (`favorite` = ⊤, `large` = ⊥) using the IALW algorithm introduced in Section 7.

3. DF-PLP

Sato's distribution semantics [Sato, 1995] start from a probability measure over a countable set of facts $\mathcal{F}$, the so-called *basic distribution*, and extends this to a probability measure over the Herbrand interpretations of the full program. It is worth noting that the basic distribution is defined independently of the logical rules and that the random variables are mutually marginally independent.

In our case, the set $\mathcal{F}$ consists of ground Boolean comparison atoms over the random variables, for which we drop the mutual marginal independence assumption. These comparison atoms form an interface between the random variables (represented as terms) and the logical layer (clauses) that reasons about truth values of atoms. While Gutmann et al. [2011] used the same principle to define the distribution semantics for Distributional Clauses, they did not support negation. [Nitti et al., 2016] extended the fixed point semantics for hybrid probabilistic logic programs (also introduced by Gutmann et al. [2011]) to stratified programs with negation . However, by doing so Nitti et al. [2016] introduced a procedural element to the semantics.

In this section we introduce the syntax and declarative semantics of DF-PLP– a probabilistic programming language with a minimal set of built-in predicates and functors. We do this in three steps. Firstly, we discuss distributional facts and the probability measure over random variables they define (Section 3.1). Secondly, we introduce the Boolean comparison atoms that form the interface layer between random variables and a logic program (Section 3.2). Thirdly, we add the logic program itself (Section 3.3). An overview table of the notation related to semantics is provided in Appendix B.

Definition 3.1 (Reserved Vocabulary)**.** *We use the following* reserved *vocabulary (built-ins), whose intended meaning is fixed across programs:*

- *a finite set Δ of* distribution functors*.*
- *a finite set Φ of* arithmetic functors*.*
- *A finite set Π of binary* comparison predicates*,*
- *the binary predicate* ~/2 *(in infix notation).*

Examples of distribution functors that we have already seen in Section 2 are `poisson`/1 and `flip`/1 but may also include further functors such as `normal`/2 to denote normal distributions. Possible arithmetic functors are `*`/2 (cf. Example 2.1) but also `max`/2, +/2, `abs`/1, etc.. Binary comparison predicates (in Prolog syntax and infix notation) are </2, >/2, =</2, >=/2, `=:=`/2, =\=/2. The precise definitions of Δ, Φ and Π are left to system designers implementing the language.

Definition 3.2 (Regular Vocabulary)**.** *We call an atom $\mu(\rho_1, \ldots, \rho_k)$ whose predicate μ/k is not part of the reserved vocabulary a* regular *atom. The set of all regular atoms constitutes the regular vocabulary.*

Note that the arguments of a predicate μ/k can contain element of $\mathcal{D}$ and $\mathcal{F}$. In this case they will have a purely logical meaning. We discuss this in more detail in Definition 4.19.

As a brief comment on notation: in the remainder of the paper we will usually denote logic program expressions in teletype font (e.g. `4>x`) when giving examples. When defining new concepts or stating theorems and propositions, we will use the Greek alphabet.

3.1. Distributional Facts and Random Variables

Definition 3.3 (Distributional Fact)**.** *A distributional fact is of the form $\nu \sim \delta$, with ν a regular ground term, and δ a ground term whose functor is in Δ. The distributional fact states that the ground term ν is interpreted as a random variable distributed according to δ.*

Definition 3.4 (Sample Space)**.** *Let ν be be a random variable distributed according to δ. The set of possible samples (or values) for ν is the sample space denoted by Ω_ν and which is determined by δ. We denote a sample from Ω_ν by $\omega(\nu)$, where ω is the* sampling *or* value *function.*

Definition 3.5 (Distributional Database)**.** *A* distributional database *is a countable set $\mathcal{D} = \{\nu_1 \sim \delta_1, \nu_2 \sim \delta_2, \dots\}$ of distributional facts, with distinct ν_i. We let $\mathcal{V} = \{\nu_1, \nu_2, \dots\}$ denote the set of random variables.*

Example 3.6. *The following distributional database encodes a Bayesian network with normally distributed random variables, two of which serve as parameters in the distribution of another one. We thus have $\mathcal{V} = \{\mathtt{x}, \mathtt{y}, \mathtt{z}\}$.*

```
% distributional facts 𝒟
x ~ normal(5,2).
y ~ normal(x,7).
z ~ normal(y,1).
```

In order for a distributional database $\mathcal{D}$ to be meaningful, it has to encode a unique joint distribution over the variables $\mathcal{V}$. The key idea is to view the set of random variables as the nodes of a Bayesian network, where each node's distribution is parameterized by the node's parents.

Definition 3.7 (Parent, Ancestor)**.** *Let $\mathcal{D}$ be a distributional database. For facts $\nu_p \sim \delta_p$ and $\nu_c \sim \delta_c$ in $\mathcal{D}$. The random variable ν_p is a* parent *of the child random variable ν_c if and only if ν_p appears in δ_c. We define* ancestor *to be the transitive closure of* parent*. A node's ancestor set is the set of all its ancestors.*

Example 3.8 (Ancestor Set)**.** *We graphically depict the ancestor set of the distributional database in Example 3.6 in Figure 3.1.*

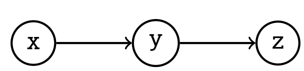

Figure 3.1: Directed acyclic graph representing the ancestor relationship between the random variables in Example 3.6. The ancestor set of `x` is the empty set, the one of `y` is $\{\mathtt{x}\}$ and the one of `z` is $\{\mathtt{x}, \mathtt{y}\}$.

Definition 3.10 (Well-Defined Distributional Database)**.** *A distributional database $\mathcal{D}$ is called* well-defined *if and only if it satisfies the following criteria:*

W1 *Each $v \in \mathcal{V}$ has a finite set of ancestors.*

W2 *The ancestor relation on the variables $\mathcal{V}$ is acyclic.*

W3 *If $v \sim \delta \in \mathcal{D}$ and the parents of v are $\{v_1, \ldots, v_m\}$, then replacing each occurrence of v_i in δ by a sample $\omega(v_i)$ always results in a well-defined distribution for v.*

The distributional database in Example 3.6 is well-defined: the ancestor relation is acyclic and finite, and as normally distributed random variables are real-valued, using such a variable as the mean of another normal distribution is always well-defined. The database would no longer be well-defined after adding `w ~ poisson(x)`, as not all real numbers can be used as a parameter of a Poisson distribution.

Definition 3.11. *A* value assignment *$\omega(\mathcal{V})$ is a combined value assignment to all random variables $\mathcal{V} = \{v_1, v_2, ...\}$, i.e., $\omega(\mathcal{V}) = (\omega(v_1), \omega(v_2), \ldots)$.*

Proposition 3.12. *A well-defined distributional database $\mathcal{D}$ defines a unique probability measure $P_{\mathcal{V}}$ on value assignments $\omega(\mathcal{V})$.*

Proof. See Appendix C.1. □

3.2. Boolean Comparison Atoms over Random Variables

Starting from the distribution over random variables defined by a well-defined distributional database, we now introduce the corresponding distribution over Boolean comparison atoms, which corresponds to the basic (discrete) distribution in Sato's distribution semantics.

Definition 3.13 (Boolean Comparison Atoms)**.** *Let $\mathcal{D}$ be a well-defined distributional database. A binary* comparison atom *$\gamma_1 \bowtie \gamma_2$ over $\mathcal{D}$ is a ground atom with predicate $\bowtie \in \Pi$. The ground terms γ_1 and γ_2 are either random variables in $\mathcal{V}$ or terms whose functor is in Φ. We denote by $\mathcal{F}$ the set of all* Lebesgue-measurable *Boolean comparison atoms over $\mathcal{D}$.*

Example 3.14. *Examples of Boolean comparison atoms over the distributional database of Example 3.6 include* `z>10`, `x<y`, `abs(x-y)=<1`, *and* `7*x=:=y+5`.

Proposition 3.15. *The probability measure $P_{\mathcal{V}}$, defined by a well-defined distributional database $\mathcal{D}$, induces a unique probability measure $P_{\mathcal{F}}$ over value assignments to the comparison atoms $\mathcal{F}$.*

Proof. See Appendix C.2. □

3.3. Logical Consequences of Boolean Comparisons

We now define the semantics of a DF-PLP program, i.e., extend the basic distribution $P_{\mathcal{F}}$ over the comparison atoms of a distributional database to a distribution over the Herbrand interpretations of a logic program using said database.

Definition 3.16 (DF-PLP Program)**.** *A DF-PLP program $\mathcal{P}^{DF} = \mathcal{D} \cup \mathcal{R}$ consists of a well-defined distributional database $\mathcal{D}$ (Definition 3.10), comparison atoms $\mathcal{F}$ (Definition 3.13), and a normal logic program $\mathcal{R}$ where clause heads belong to the regular vocabulary (cf. Definition 3.2), and which can use comparison atoms from $\mathcal{F}$ in their bodies.*

Example 3.17. *We further extend the running example.*

```
% distributional facts 𝒟
x ~ normal(5,2).
y ~ normal(x,7).
z ~ normal(y,1).
% logic program ℛ
a :- abs(x-y) =< 1.
b :- not a, z>10.
```

The logic program defines two logical consequences of Boolean comparisons over the Bayesian network, where `a` *is true if the absolute difference between random variables* `x` *and* `y` *is at most one, and* `b` *is true if* `a` *is false, and the random variable* `z` *is greater than* 10*.*

In order to extend the basic distribution to logical consequences, i.e. logical rules, we require the notion of a *consistent comparisons database* (CCD). The key idea is that samples of the random variables in $\mathcal{D}$ jointly determine a truth value assignment to the comparison atoms in $\mathcal{F}$.

Definition 3.18 (Consistent Comparisons Database)**.** *Let $\mathcal{D}$ be a well-defined distributional database, $\mathcal{F} = \{\kappa_1, \kappa_2, \ldots\}$ the corresponding set of measurable Boolean comparison atoms, and $\omega(\mathcal{V})$ a value assignment to all random variables $\mathcal{V} = \{v_1, v_2, ...\}$. We define $I_{\omega(\mathcal{V})}(\kappa_i) = \top$ if κ_i is true after setting all random variables to their values under $\omega(\mathcal{V})$, and $I_{\omega(\mathcal{V})}(\kappa_i) = \bot$ otherwise. $I_{\omega(\mathcal{V})}$ induces the consistent comparisons database $\mathcal{F}_{\omega(\mathcal{V})} = \{\kappa_i \mid I_{\omega(\mathcal{V})}(\kappa_i) = \top\}$.*

To define the semantics of a DF-PLP program $\mathcal{P}^{DF}$, we now require that, given a CCD $\mathcal{F}_{\omega(\mathcal{V})}$, the logical consequences in $\mathcal{P}^{DF}$ are uniquely defined. As common in the PLP literature, we achieve this by requiring the program to have a two-valued well-founded model [Van Gelder et al., 1991] for each possible value assignment $\omega(\mathcal{V})$.

Definition 3.19 (Valid DF-PLP Program)**.** *A DF-PLP program $\mathcal{P}^{DF} = \mathcal{D} \cup \mathcal{R}$ is called* valid *if and only if for each CCD $\mathcal{F}_{\omega(\mathcal{V})}$, the logic program $\mathcal{F}_{\omega(\mathcal{V})} \cup \mathcal{R}$ has a two-valued well-founded model.*

We follow the common practice of defining the semantics with respect to ground programs; the semantics of a program with non-ground $\mathcal{R}$ is defined as the semantics of its grounding with respect to the Herbrand universe.

Proposition 3.20. *A valid DF-PLP program $\mathcal{P}^{DF}$ induces a unique probability measure $P_{\mathcal{P}^{DF}}$ over Herbrand interpretations.*

Proof. See Appendix C.3. □

Definition 3.21. *We define the declarative semantics of a DF-PLP program $\mathcal{P}^{DF}$ to be the probability measure $P_{\mathcal{P}^{DF}}$.*

In contrast to the imperative semantics of Nitti et al. [2016], in DF-PLP the connection between comparison atoms and the logic program is purely declarative. That is, logic program negation on comparison atoms negates the (interpreted) comparison. For example, if we have a random variable `n`, then `n>=2` is equivalent to `not n<2`. Such equivalences do not hold in the stratified programs introduced by Nitti et al. [2016]. This then allows the programmer to refactor the logic part as one would expect.

4. DC-ProbLog

While the previous section has focused on the core elements of the DC-ProbLog language, we now introduce syntactic sugar to ease modeling. We consider three kinds of modeling patterns in DF-PLP, and introduce a more compact notation for each of them. We focus on examples and intuitions first. Subsequently, we formally define the semantics of DC-ProbLog (DF-PLP + syntactic sugar) in Section 4.2.

4.1. Syntactic Sugar: Syntax and Examples

4.1.1. Boolean Random Variables

The first modelling pattern concerns Boolean random variables, which we already encountered in Section 2.1 as probabilistic facts (in DC-ProbLog) or as a combination of a Bernoulli random variable, a comparison atom, and a logic rule (in DF-PLP). Below we give a more concise example.

Example 4.1. *We model, in DF-PLP, an alarm that goes off for different reasons.*

```
issue1 ~ flip(0.1).
issue2 ~ flip(0.6).
issue3 ~ flip(0.3).

alarm :- issue1=:=1, not issue2=:=1.
alarm :- issue3=:=1, issue1=:=0.
alarm :- issue2=:=1.
```

To make such programs more readable, we borrow the well-known notion of *probabilistic fact* from discrete PLP, which directly introduces a logical atom as alias for the comparison of a random variable with the value `1`, together with the probability of that value being taken.

Definition 4.2 (Probabilistic Fact)**.** *A probabilistic fact is of the form $p::\mu$, where p is an arithmetic term that evaluates to a real number in the interval $[0,1]$ and μ is a regular ground atom.*

Example 4.3. *We use probabilistic facts to rewrite the previous example.*

```
0.1::problem1.
0.6::problem2.
0.3::problem3.

alarm :- problem1, not problem2.
alarm :- problem3, not problem1.
alarm :- problem2.
```

4.1.2. Probabilistically Selected Logical Consequences

The second pattern concerns situations where a random variable with a finite domain is used to model a choice between several logical consequences:

Example 4.4. *We use a random variable to model a choice between whom to call upon hearing the alarm.*

```
call ~ finite([0.6:1,0.2:2,0.1:3]).
alarm.
call(mary) :- call=:=1, alarm.
call(john) :- call=:=2, alarm.
call(police) :- call=:=3, alarm.
```

To more compactly specify such situations, we borrow the concept of an *annotated disjunction* from discrete PLP [Vennekens et al., 2004].

Definition 4.5 (Annotated Disjunction)**.** *An annotated disjunction (AD) is a rule of the form*

$$p_1 :: \mu_1; \ldots ; p_n :: \mu_n \text{:-} \beta.$$

where the p_i's are arithmetic terms each evaluating to a number in $[0, 1]$ *with a total sum of at most* 1*. The μ)i's are regular gorund atoms, and β is a (possibly empty) conjunction of literals.*

The informal meaning of such an AD is "if β is true, it probabilistically causes one of the μ_i (or none of them, if the probabilities sum to less than one) to be true as well".

Example 4.6. *We now use an AD for the previous example.*

```
alarm.
0.6::call(mary); 0.2::call(john); 0.1::call(police) :- alarm.
```

It is worth noting that the same head atom may appear in multiple ADs, whose bodies may be non-exclusive, i.e., be true at the same time. That is, while a single AD *can* be used to model a multi-valued random variable, *not all* ADs encode such variables.

Example 4.7. *The following program models the effect of two kids throwing stones at a window.*

```
0.5::throws(suzy).
throws(billy).

0.8::effect(broken); 0.2::effect(none) :- throws(suzy).
0.6::effect(broken); 0.4::effect(none) :- throws(billy).
```

Here, we have $P($`effect(broken)`*$) = 0.76$ and $P($*`effect(none)`*$) = 0.46$, as there are worlds where both* `effect(broken)` *and* `effect(none)` *hold. The two ADs do hence not encode a categorical distribution. This is explicit in the DF-PLP program, which contains two random variables (*`x1` *and* `x2`*):*

```
x0 ~ flip(0.5).
throws(suzy) :- x0=:=1.
throws(billy).

x1 ~ finite([0.8:1,0.2:2]).
effect(broken) :- x1=:=1, throws(suzy).
effect(none) :- x1=:=2, throws(suzy).
x2 ~ finite([0.6:1,0.4:2]).
effect(broken) :- x2=:=1, throws(billy).
effect(none) :- x2=:=2, throws(billy).
```

4.1.3. Context-Dependent Distributions

The third pattern is concerned with situations where the same conclusion is based on random variables with different distributions depending on specific properties of the situation, as illustrated by the following example.

Example 4.8. *We use two separate random variables to model that whether a machine works depends on the temperature being below or above a threshold. The temperature follows different distributions based on whether it is a hot day or not, but the threshold is independent of the type of day.*

```
0.2::hot.

temp_hot ~ normal(27,5).
temp_not_hot ~ normal(20,5).

works :- hot, temp_hot<25.0.
works :- not hot, temp_not_hot<25.0.
```

To more compactly specify such situations, we borrow the syntax of *distributional clauses* from the DC language [Gutmann et al., 2011], which we already encountered in Section 2.1.

Definition 4.9 (Distributional Clause)**.** *A distributional clause (DC) is a rule of the form*

$$\tau \sim \delta :\!- \beta.$$

where τ is a regular ground expression, the functor of δ is in Δ, and β is a conjunction of literals.

We call the left-hand side of the $\sim/2$ prediate in a distributional clause a *random term* and the right-hand side a *distribution term*. Note that random terms cannot always be interpreted as random variables, which we discuss now.

The informal meaning of a distributional clause is "if β is true, then the random term τ refers to a random variable that follows a distribution given by the distribution term δ". Here, the distinction between *refers to* a random variable and *is* a random variable becomes crucial, as we will often have several distributional clauses for the same random term. This is also the case in the following example.

Example 4.10. *Using distributional clauses, we can rewrite the previous example with a single random term* `temp` *as*

```
0.2::hot.

temp ~ normal(27,5) :- hot.
temp ~ normal(20,5) :- not hot.

works :- temp < 25.0.
```

The idea is that we still have two underlying random variables, one for each distribution, but the logic program uses the same term to refer to both of them depending on the logical context. The actual comparison facts are on the level of these implicit random variables, and `temp<0.25` *refers to one of them depending on context, just as in the original example.*

4.2. Syntactic Sugar: Semantics

We now formalize the declarative semantics of DC-ProbLog, i.e. DF-PLP extended with probabilistic facts, annotated disjunctions and distributional clauses, The idea is to define program transformations that eliminate these three modelling constructs from a DC-ProbLog program, resulting in a DF-PLP program for which we have defined the semantics in Section 3.

Throughout this section, we will treat distributional facts as distributional clauses with empty bodies, and we will only consider ground programs for ease of notation. As usual, a non-ground program is shorthand for its Herbrand grounding.

Definition 4.11 (Statement)**.** *A* DC-ProbLog statement *is either a probabilistic fact, an annotated disjunction, a distributional clause, or a normal clause.*

Definition 4.12 (DC-ProbLog program)**.** *A DC-ProbLog program $\mathcal{P}$ is a countable set of ground DC-ProbLog statements.*

4.2.1. Eliminating Probabilistic Facts and Annotated Disjunctions

Example 4.13. *We use the following DC-ProbLog program as running example.*

```
p ~ beta(1,1).
p::a.
b ~ normal(3,1) :- a.
b ~ normal(10,1) :- not a.
```

```
c ~ normal(b,5).
0.2::d; 0.5::e; 0.3::f :- not b<5, b < 10.
g :- a, not f, b+c<15.
```

Definition 4.14 (Eliminating Probabilistic Facts and ADs)**.** *Let $\mathcal{P}$ be a DC-ProbLog program. We define the following transformation rules to eliminate probabilistic facts and annotated disjunctions.*

- *Replace each probabilistic fact $p :: \mu$ in $\mathcal{P}$ by*

$$\begin{aligned}&\nu \sim flip(p).\\ &\mu :\!- \nu =:= 1.\end{aligned}$$

  with a fresh random variable ν for each probabilstic fact.

- *Replace each AD $p_1 :: \mu_1; \ldots; p_n :: \mu_n :\!- \beta$ in $\mathcal{P}$ by*

$$\begin{aligned}&\nu \sim finite([p_1 : 1, \ldots, p_n : n])\\ &\mu_1 :\!- \nu =:= 1, \beta.\\ &\ldots\\ &\mu_n :\!- \nu =:= n, \beta.\end{aligned}$$

  with a fresh random variable ν for each AD.

Note that if the probability label(s) of a fact or AD include random terms, as in the case of `p::a` in the Example 4.13, then these are parents of the newly introduced random variable. However, the new random variable will not be a parent of other random variables, as they are only used locally within the new fragments. They thus introduce neither cycles nor infinite ancestor sets into the program.

Definition 4.15 (AD-Free Program)**.** *An* AD-free *DC-ProbLog program $\mathcal{P}^*$ is a DC-ProbLog program that contains neither probabilistic facts nor annotated disjunctions. We denote by $\mathcal{H}_{\mathcal{P}^*}$ the set of atoms $\tau \sim \delta$ that appear as head of a distributional clause in $\mathcal{P}^*$, and by $\mathcal{T}_{\mathcal{P}^*}$ the set of random terms in $\mathcal{H}_{\mathcal{P}^*}$.*

Example 4.16. *Applying Definition 4.14 to Example 4.13 results in*

```
p ~ beta(1,1).
x ~ flip(p).
a :- x =:= 1.
b ~ normal(3,1) :- a.
b ~ normal(10,1) :- not a.
c ~ normal(b,5).
y ~ finite([0.2:1,0.5:2,0.3:3]).
d :- y =:= 1, not b<5, b < 10.
e :- y =:= 2, not b<5, b < 10.
f :- y =:= 3, not b<5, b < 10.
g :- a, not f, b+c<15.
```

We have $\mathcal{H}_{\mathcal{P}^*}$ = { `p~beta(1,1)`, `x~flip(p)`, `b~normal(3,1)`, `b~normal(10,1)`, `c~normal(b,5)`, `y~finite[0.2:1,0.5:2.0.3:3])` }. *Furthermore, we also have* $\mathcal{T}_{\mathcal{P}^*} = \{\mathrm{p}, \mathrm{x}, \mathrm{b}, \mathrm{c}, \mathrm{y}\}$.

4.2.2. Eliminating Distributional Clauses

While eliminating probabilistic facts and annotated disjunctions is a rather straightforward local transformation, eliminating distributional clauses is more involved. The reason is that a distributional clause has a global effect in the program, as it defines a condition under which a random term has to be *interpreted* as a specific random variable when mentioned in a distributional clause or comparison atom. Therefore, eliminating a distributional clause involves both introducing the relevant random variable explicitly to the program and pushing the condition from the body of the distributional clause to all the places in the logic program that interpret the original random term.

Before delving into the mapping from an AD-free DC-ProbLog to a DF-PLP program, we introduce some relevant terminology.

Definition 4.17 (Parent, Ancestor)**.** *Given an AD-free program $\mathcal{P}^*$ with τ_p and τ_c in $\mathcal{T}_{\mathcal{P}^*}$. We call τ_p a* parent *of τ_c if and only if τ_p appears in the distribution term δ_c associated with τ_c in $\mathcal{H}_{\mathcal{P}^*}$ ($\tau_c \sim \delta_c \in \mathcal{H}_{\mathcal{P}^*}$). We define* ancestor *to be the transitive closure of* parent*.*

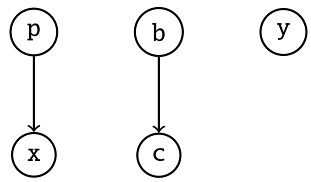

Figure 4.1: Directed acyclic graph representing the ancestor relationship between the random variables in Example 4.15. The random terms `p`, `b` and `y` have the empty set as their ancestor set. The ancestor set of `x` is {`p`} and `c` is {`b`}.

For random terms, we distinguish *interpreted occurrences* of the term that need to be resolved to the correct random variable from other occurrences where the random term is treated as any other term in a logic program, e.g., as an argument of a logical atom.

Definition 4.19 (Interpreted Occurrence)**.** *An* interpreted occurrence *of a random term τ in an AD-free program $\mathcal{P}^*$ is one of the following:*

- *the use of τ as parameter of a distribution term in the head of a distributional clause in $\mathcal{P}^*$*
- *the use of τ in a comparison literal in the body of a (distributional or normal) clause in $\mathcal{P}^*$*

We say that a clause interprets *τ if there is at least one interpreted occurrence of τ in the clause.*

Definition 4.20 (Well-Defined AD-free Program)**.** *Given an AD-free program $\mathcal{P}^*$ with $C_{\mathcal{P}^*}$ the set of distributional clauses in $\mathcal{P}^*$, we call $C_{\mathcal{P}^*}$* well-defined *if the following conditions hold:*

DC1 *For each random term $\tau \in \mathcal{T}_{\mathcal{P}^*}$, the number of distributional clauses $\tau \sim \delta :\!- \beta$ in $\mathcal{P}^*$ is finite, and these clauses all have mutually exclusive bodies. This means that only a single rule can be true at once.*

DC2 *All distribution terms in $C_{\mathcal{P}^*}$ are well-defined for all possible values of the random terms they interpret.*

DC3 *Each random term has a finite set of ancestors.*

DC4 *The ancestor relation is acyclic.*

We now discuss how to reduce a (valid) DC-ProbLog program to a DF-PLP program. This happens in two steps. First, we eliminate distributional clauses and introduce appropriate distributional facts instead (see Definition 4.21). Second, we *contextualize* interpreted occurrences of random terms in clause bodies (see Definition 4.22).

The first step introduces a new built-in predicate rv/2 that associates random terms in a well-defined AD-free program with explicit random variables in the DF-PLP program it is transformed into. This predicate is used in the bodies of clauses that interpret random terms (cf. Definition 4.19) to appropriately contextualize those.

The idea behind the built-in rv/2 predicate is to restrict the applicability of a clause to contexts where all the random terms can be interpreted, i.e. to contexts where the random terms are random variables. This implies that in contexts where such a random term cannot be interpreted, the *entire* body evaluates to false.

Definition 4.21 (Eliminating Distributional Clauses)**.** *Let $C_{\mathcal{P}^*}$ be a well-defined set of distributional clauses. We denote by $\delta_{\rho_1,\dots,\rho_k}$ a distribution term that involves exactly k different random terms $\rho_1, \dots, \rho_k$. For each ground random term $\tau \in \mathcal{T}_{\mathcal{P}^*}$ we simultaneously define the following sets:*

- *the set of distributional facts for τ*

$$\mathcal{D}(\tau) = \{\tau^{\beta}_{\nu_1,\dots,\nu_k} \sim \delta^{\beta}_{\nu_1,\dots,\nu_k} \mid (\tau \sim \delta_{\rho_1,\dots,\rho_k} :\!- \beta \in C_{\mathcal{P}^*}, \nu_1 \in \mathcal{V}(\rho_1), \dots, \nu_k \in \mathcal{V}(\rho_k)\}$$

- *the set of (fresh) random variables for τ*

$$\mathcal{V}(\tau) = \{\nu \mid \nu \sim \delta \in \mathcal{D}(\tau)\}$$

- *the set of context clauses for τ*

$$\mathcal{R}^c(\tau) = \Big\{\mathtt{rv}(\tau, \tau^{\beta}_{\nu_1,\dots,\nu_k}) :\!- \mathtt{rv}(\rho_1, \nu_1), \dots, \mathtt{rv}(\rho_k, \nu_k), \beta \;\Big|\; \tau \sim \delta_{\rho_1,\dots,\rho_k} :\!- \beta \in C_{\mathcal{P}^*}, \nu_1 \in \mathcal{V}(\rho_1), \dots, \nu_k \in \mathcal{V}(\rho_k)\Big\}$$

At first glance, Definition 4.21 seems to contain a mutual recursion involving $\mathcal{D}(\cdot)$ and $\mathcal{V}(\cdot)$. However, if we recall that for a well-defined set of distributional clauses

$C_{\mathcal{P}^*}$ the ancestor relationship between random terms constitutes an acyclic directed graph, the apparent mutual recursion evaporates. We can now define the distributional facts encoding of the distributional clauses, which will give rise to a DF-PLP program instead of DC-ProbLog program.

Definition 4.22 (Distributional Facts Encoding)**.** *Let $\mathcal{P}^*$ be an AD-free DC-ProbLog program and $C_{\mathcal{P}^*}$ its set of distributional clauses. We define the distributional facts encoding of $C_{\mathcal{P}^*}$ as $C^{DF}_{\mathcal{P}^*} := \mathcal{D} \cup \mathcal{R}^c$, with*

$$\mathcal{D} = \bigcup_{\tau \in \mathcal{T}_{\mathcal{P}^*}} \mathcal{D}(\tau) \qquad\qquad \mathcal{R}^c = \bigcup_{\tau \in \mathcal{T}_{\mathcal{P}^*}} \mathcal{R}^c(\tau)$$

using $\mathcal{D}(\cdot)$ and $\mathcal{R}^c(\cdot)$ from Definition 4.21.

Example 4.23 (Eliminating Distributional Clauses)**.** *We demonstrate the elimination of distributional clauses using the DCs in Example 4.16, i.e.*

```
1 p ~ beta(1,1).
2 x ~ flip(p).
3 b ~ normal(3,1) :- a.
4 b ~ normal(10,1) :- not a.
5 c ~ normal(b,5).
6 y ~ finite([0.2:1,0.5:2,0.3:3]).
```

Here, the distribution terms in Line 2 and Line 5 (`flip(p)` and `normal(b,5)`) contain one parent random term each (`p` and `b`, respectively), whereas all others have no parents. As `b` is defined by two clauses, we get fresh random variables for each of them, which in turn introduces different fresh random variables for the child `c`. This gives us:

```
v1 ~ beta(1,1).
rv(p,v1).
v2 ~ flip(v1).
rv(x,v2) :- rv(p,v1).
v3 ~ normal(3,1).
rv(b,v3) :- a.
v4 ~ normal(10,1).
rv(b,v4) :- not a.
v5 ~ normal(v3,5).
rv(c,v5) :- rv(b,v3).
v6 ~ normal(v4,5).
rv(c,v6) :- rv(b,v4).
v7 ~ finite([0.2:1,0.5:2,0.3:3]).
rv(y,v7).
```

Eliminating distributional clauses (following Definition 4.21) introduces the distributional facts and context rules necessary to encode the original distributional clauses. To complete the transformation to a DF-PLP program, we further transform the logical rules. Prior to that, however, we need to define the *contextualization function*.

Definition 4.24 (Contextualization Function)**.** *Let β be a conjunction of atoms and let its comparison literals interpret the random terms $\tau_1, \ldots, \tau_n$. Furthermore, let Λ_i be a special logical variable associated to a random term $\tau_i \in \mathcal{T}_{\mathcal{P}^*}$ for each τ_i. We define $K(\beta)$ to be the conjunction of literals obtained by replacing the interpreted occurrences of the τ_i in β by their corresponding Λ_i and conjoining to this modified conjunction* `rv`(τ_i, Λ_i) *for each τ_i. We call $K(\cdot)$ the contextualization function.*

Definition 4.25 (Contextualized Rules)**.** *Let $\mathcal{P}^*$ be an AD-free program with logical rules $\mathcal{R}^{\mathcal{P}^*}$ and distributional clauses $C_{\mathcal{P}^*}$, and let $C_{\mathcal{P}^*}^{DF} = \mathcal{D} \cup \mathcal{R}^c$ be the distributional facts encoding of $C_{\mathcal{P}^*}$. We define the contextualization of the bodies of the rules $\mathcal{R}^{\mathcal{P}^*} \cup \mathcal{R}^{DF}$ as a two-step process:*

a. Apply the contextualization function K to all bodies in $\mathcal{R}^{\mathcal{P}^*} \cup \mathcal{R}^c$ and obtain:

$$\mathcal{R}^{\Lambda} = \{\eta :- K(\beta) \mid \eta :- \beta \in \mathcal{R}^{\mathcal{P}^*} \cup \mathcal{R}^c\}$$

b. Obtain the set of ground logical rules $\mathcal{R}$ by grounding each logical variable Λ_i in $\mathcal{R}^{\Lambda}$ with random variables $\nu_i \in \mathcal{V}(\tau_i)$ in all possible ways.

We call $\mathcal{R}$ the contextualized logic program of $\mathcal{P}^*$.

The contextualization function $K(\cdot)$ creates non-ground comparison atoms, e.g. `L>5`. Contrary to (ground) random terms, non-ground logical variables in such a comparison atom are not interpreted occurrences (cf. Definition 4.19) and the comparison itself only has a logical meaning. By grounding out the freshly introduced logical variables we obtain a purely logical program where the comparison atoms contain either arithmetic expressions or random variables (instead of random terms).

Example 4.26 (Contextualizing Random Terms)**.** *Let us now study the effect of the second transformation step. Consider again the AD-free program in Example 4.16 and the set of rules and distributional clauses obtained in Example 4.23. The contextualization step T2a rewrites the logical rules in the AD-free input program to*

```
a :- rv(x,Lx), Lx =:= 1.
d :- rv(y,Ly), rv(b,Lb), Ly =:= 1, not Lb<5, Lb < 10.
e :- rv(y,Ly), rv(b,Lb), Ly =:= 2, not Lb<5, Lb < 10.
f :- rv(y,Ly), rv(b,Lb), Ly =:= 3, not Lb<5, Lb < 10.
g :- rv(b,Lb), rv(c,Lc), a, not f, Lb+Lc < 15.
```

These rules then get instantiated (step T2b) to

```
a :- rv(x, v2), v2 =:= 1.
d :- rv(y,v7), rv(b,v3), v7 =:= 1, not v3<5, v3 < 10.
e :- rv(y,v7), rv(b,v3), v7 =:= 2, not v3<5, v3 < 10.
f :- rv(y,v7), rv(b,v3), v7 =:= 3, not v3<5, v3 < 10.
d :- rv(y,v7), rv(b,v4), v7 =:= 1, not v4<5, v4 < 10.
e :- rv(y,v7), rv(b,v4), v7 =:= 2, not v4<5, v4 < 10.
f :- rv(y,v7), rv(b,v4), v7 =:= 3, not v4<5, v4 < 10.
g :- rv(b,v3), rv(c,v5), a, not f, v3+v5<15.
```

```
g :- rv(b,v3), rv(c,v6), a, not f, v3+v6<15.
g :- rv(b,v4), rv(c,v5), a, not f, v4+v5<15.
g :- rv(b,v4), rv(c,v6), a, not f, v4+v6<15.
```

Together with the distributional facts and rules obtained in Example 4.23, this last block of rules forms the DC-PLP program that specifies the semantics of the AD-free DC-ProbLog program, and thus the semantics of the DC-ProbLog program in Example 4.13.

We note that the mapping from an AD-free program to a set of distributional facts and contextualized rules as defined here is purely syntactical, and written to avoid case distinctions. Therefore, it usually produces overly verbose programs. For instance, for random terms introduced by a distributional fact, the indirection via `rv` is only needed if there is a parent term in the distribution that has context-specific interpretations. The grounding step may introduce rule instances whose conjunction of `rv`-atoms is inconsistent. This is for example the case for the last three rules for `g` in the Example 4.26, which we illustrate in the example below.

Example 4.27. *The following is a (manually) simplified version of the DF-PLP program for the running example, where we propagated definitions of* `rv`*-atoms:*

```
v1 ~ beta(1,1).
v2 ~ flip(v1).
v3 ~ normal(3,1).
v4 ~ normal(10,1).
v5 ~ normal(v3,5).
v6 ~ normal(v4,5).
v7 ~ finite([0.2:1,0.5:2,0.3:3]).

a :- v2 =:= 1.
d :- a,     v7 =:= 1, not v3<5, v3 < 10.
e :- a,     v7 =:= 2, not v3<5, v3 < 10.
f :- a,     v7 =:= 3, not v3<5, v3 < 10.
d :- not a, v7 =:= 1, not v4<5, v4 < 10.
e :- not a, v7 =:= 2, not v4<5, v4 < 10.
f :- not a, v7 =:= 3, not v4<5, v4 < 10.
g :- a,     a,     a, not f, v3+v5<15.
g :- a,     not a, a, not f, v3+v6<15.  % inconsistent
g :- not a, a,     a, not f, v4+v5<15.  % inconsistent
g :- not a, not a, a, not f, v4+v6<15.  % inconsistent
```

In the bodies of the last three rules we have, inter alia, conjunctions of `a` *and* `not a`*. This can never be satisfied and renders the bodies of these rules inconsistent.*

Definition 4.28 (Semantics of AD-free DC-ProbLog Programs)**.** *The semantics of an AD-free DC-ProbLog program $\mathcal{P}^*$ is the semantics of the DF-PLP program $\mathcal{P}^{DF,*} = \mathcal{D} \cup \mathcal{R}$. We call $\mathcal{P}^*$ valid if and only if $\mathcal{P}^{DF,*}$ is valid.*

Definition 4.29 (Semantics of $DC - ProbLog$ Programs)**.** *The semantics of a $DC -$ $ProbLog$ program $\mathcal{P}$ is the semantics of the AD-free DC-ProbLog program $\mathcal{P}^*$. We call $\mathcal{P}$ valid if and only if $\mathcal{P}^*$ is valid.*

Programs with distributional clauses can make programs with combinatorial structures more readable by grouping random variables with the same role under the same random term. However, the programmer needs to be aware of the fact that distributional clauses have non-local effects on the program, as they affect the interpretation of their random terms also outside the distributional clause itself. This can be rather subtle, especially if the bodies of the distributional clauses with the same random term are not exhaustive. We discuss this issue in more detail in Appendix D.

4.3. Syntactic Sugar: Validity

As stated above, a DC-ProbLog program $\mathcal{P}$ is syntactic sugar for an AD-free program $\mathcal{P}^*$ (Definition 4.14), and is valid if $\mathcal{P}^{DF,*}$ as specified in Definition 4.28 is a valid DF-PLP program, i.e. the distributional database is well-defined, the comparison literals are measurable, and each consistent fact database results in a two-valued well-founded model if added to the logic program (Definition 3.19). For the distributional database to be well-defined (Definition 3.10), it suffices to have $\mathcal{C}_{\mathcal{P}^*}$ well-defined (Definition 4.20), as can be verified by comparing the relevant definitions. Indeed, a well-defined $\mathcal{C}_{\mathcal{P}^*}$ is a precondition for the transformation as stated in the definition.

The transformation changes neither distribution terms nor comparison literals, and thus maintains measurability of the latter. As far as the logic program structure is concerned, the transformation to a DF-PLP adds rules for `rv` based on the bodies of all distributional clauses, and uses positive `rv` atoms in the bodies of all clauses that interpret random terms to ensure that all interpretations of random variables are anchored in the appropriate parts of the distributional database. This level of indirection does not affect the logical reasoning for programs that only interpret random terms in appropriate contexts. It is the responsibility of the programmer to ensure that this is the case and indeed results in appropriately defined models.

4.4. Syntactic Sugar: Additional Constructs

4.4.1. User-Defined Sample Spaces

The semantics of DC-ProbLog as presented in the previous sections only alllows for random variables with numerical sample spaces, e.g. normal distributions, or Poisson distributions. For categorical random variables, however, one might like to give a specific meaning to the elements in the sample space instead of a numerical value.

Example 4.30. *Consider the following program:*

```
color ~ uniform([r,g,b]).
red:- color=:=r.
```

Here we discribe a categorical random variable (uniformaly distributed) whose sample space is the set of expressions $\{\mathtt{r},\mathtt{b},\mathtt{g}\}$*. By simply associating a natural number to each element of the sample space we can map the program back to a program whose semantics we already defined:*

```
color ~ uniform([1,2,3]).
r:- color=:=1,
red:- r.
```

Swapping out the sample space of discrete random variables with natural numbers is always possible as the cardinality of such a sample space is either smaller (finite categorical) or equal (infinite) to the cardinality of the natural numbers.

4.4.2. Multivariate Distributions

Until now we have restricted the syntax and semantics of DC-ProbLog to univariate distributions, e.g. the univariate normal distribution. At first this might seem to severely limit the expressivity of DC-ProbLog, as probabilistic modelling with multivariate random variables is a common task in modern statistics and probabilistic programming. However, this concern is voided by realizing that multivariate random variables can be decomposed into *combinations* of independent univariate random variables. We will illustrate this on the case of the bivariate normal distribution.

Example 4.31 (Constructing the Bivariate Normal Distribution)**.** *Assume we would like to construct a random variable distributed according to a bivariate normal distribution:*

$$\begin{pmatrix} \nu_1 \\ \nu_2 \end{pmatrix} \sim \mathcal{N}\left(\begin{pmatrix} \mu_1 \\ \mu_2 \end{pmatrix}, \begin{pmatrix} \sigma_{11} & \sigma_{12} \\ \sigma_{21} & \sigma_{22} \end{pmatrix} \right)$$

The equation above can be rewritten as:

$$\begin{pmatrix} \nu_1 \\ \nu_2 \end{pmatrix} \sim \begin{pmatrix} \mu_1 \\ \mu_2 \end{pmatrix} + \begin{pmatrix} \eta_{11} & \eta_{12} \\ \eta_{21} & \eta_{22} \end{pmatrix} \begin{pmatrix} \mathcal{N}(0, \lambda_1) \\ \mathcal{N}(0, \lambda_2) \end{pmatrix}$$

where it holds that

$$\begin{pmatrix} \sigma_{11} & \sigma_{12} \\ \sigma_{21} & \sigma_{22} \end{pmatrix} = \begin{pmatrix} \eta_{11} & \eta_{12} \\ \eta_{21} & \eta_{22} \end{pmatrix} \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix} \begin{pmatrix} \eta_{11} & \eta_{21} \\ \eta_{12} & \eta_{22} \end{pmatrix}$$

It can now be shown that the bivariate distributions can be expressed as:

$$\begin{pmatrix} \nu_1 \\ \nu_2 \end{pmatrix} \sim \begin{pmatrix} \mathcal{N}(\mu_{\nu_1}, \sigma_{\nu_1}) \\ \mathcal{N}(\mu_{\nu_2}, \sigma_{\nu_2}) \end{pmatrix}$$

where μ_{ν_1}*,* μ_{ν_2}*,* σ_{ν_1} *and* σ_{ν_2} *can be expressed as:*

$$\mu_{\nu_1} = \mu_1 \qquad \sigma_{\nu_1} = \sqrt{\eta_{11}\lambda_1^2 + \eta_{12}\lambda_2^2}$$

$$\mu_{\nu_2} = \mu_2 \qquad \sigma_{\nu_2} = \sqrt{\eta_{21}\lambda_1^2 + \eta_{22}\lambda_2^2}$$

We conclude from this that a bivariate normal distribution can be modeled using two univariate normal distributions that have a shared set of parameters and is thereby semantically defined in DC-ProbLog.

Expressing multivariate random variables in a user-friendly fashion in a probabilistic programming language is simply a matter of adding syntactic sugar for combinations of univariate random variables once the semantics are defined for the latter.

Example 4.32 (Bivariate Normal Distribution)**.** *Possible syntactic sugar to declare a bivariate normal distribution in DC-ProbLog, where the mean of the distribution in the two dimensions is* 0.5 *and* 2*, and the covariance matrix is* $\begin{bmatrix} 2 & 0.5 \\ 0.5 & 1 \end{bmatrix}$.

```
(x1,x2) ~ normal2D([0.5,2], [[2, 0.5],[0.5,1]])
q:- x1<0.4, x2>1.9.
```

On the inference side, the special syntax might then additionally be used to deploy dedicated inference algorithms. This is usually done in probabilistic programming languages that cater towards inference with multivariate (and often continuous) random variables [Carpenter et al., 2017, Bingham et al., 2019]. Note that probability distributions are usually constructed by applying transformations to a set of independent uniform distribution. From this viewpoint the builtin-in `normal/2`, denoting the univariate normal distribution, is syntactic sugar for such a transformation as well.

5. Probabilistic Inference Tasks

In Section 3.3 we defined the probability distribution induced by a DF-PLP program by extending the basic distribution to logical consequences (expressed as logical rules). The joint distribution is then simply the joint distribution over all (ground) logical consequences. We obtain marginal probability distributions by marginalizing out specific logical consequences.[1] This means that marginal and joint probabilities of atoms in DF-PLP programs are well-defined. Defining the semantics of probabilistic logic programs using an extension of Sato's distribution semantics gives us the semantics of probabilistic queries: the probability of an atom of interest is given by the probability induced by the joint probability of the program and marginalizing out all atoms one is not interested in.

The situation is more involved with regard to conditional probability queries. In contrast to unconditional queries, not all conditional queries are well-defined under the distribution semantics. We will now give the formal definition of the PROB task, which lets us compute the (conditional) marginal probability of probabilistic events and which has so far not yet been defined in the PLP literature for hybrid domains under a declarative semantics (e.g. [Azzolini et al., 2021]).

After defining the task of computing conditional marginal probabilities, we will study how to compute these probabilities in the hybrid domain. Before defining the PROB task, we will first need to formally introduce the notion of a conditional probability with respect to a DC-ProbLog program.

[1] This is possible as the compatibility condition is satisfied by construction in the distribution semantics. See also the proof of Proposition 3.15 in Section C.2.

Definition 5.1 (Conditional Probability)**.** *Let $\mathcal{A}$ be the set of all ground atoms in a given DC-ProbLog program $\mathcal{P}$. Let $\mathcal{E} = \{\eta_1, \dots, \eta_n\} \subset \mathcal{A}$ be a set of observed atoms, and $e = \langle e_1, \dots, e_n \rangle$ a vector of corresponding observed truth values, with $e_i \in \{\bot, \top\}$. We refer to $(\eta_1 = e_1) \land \ldots \land (\eta_n = e_n)$ as the evidence and write more compactly $\mathcal{E} = e$. Let $\mu \in \mathcal{A}$ be an atom of interest called the query. If the probability of $\mathcal{E} = e$ is greater than zero, then the conditional probability of $\mu = \top$ given $\mathcal{E} = e$ is defined as:*

$$P_{\mathcal{P}}(\mu = \top \mid \mathcal{E} = e) = \frac{P_{\mathcal{P}}(\mu = \top, \mathcal{E} = e)}{P_{\mathcal{P}}(\mathcal{E} = e)} \tag{5.1}$$

Definition 5.2 (PROB Task)**.** *Let $\mathcal{A}$ be the set of all ground atoms of a given DC-ProbLog program $\mathcal{P}$. We are given the (potentially empty) evidence $\mathcal{E} = e$ (with $\mathcal{E} \subset \mathcal{A}$) and a set $\mathcal{Q} \subset \mathcal{A}$ of atoms of interest, called query atoms. The* **PROB task** *consists of computing the conditional probability of the truth value of every atom in $\mathcal{Q}$ given the evidence, i.e. compute the conditional probability $P_{\mathcal{P}}(\mu{=}\top \mid \mathcal{E}{=}e)$ for each $\mu \in \mathcal{Q}$.*

Example 5.3 (Valid Conditioning Set)**.** *Assume two random variables ν_1 and ν_2, where ν_1 is distributed according to a normal distribution and ν_2 is distributed according to a Poisson distribution. Furthermore, assume the following conditioning set $\mathcal{E} = \{\eta_1 = \top, \eta_2 = \top\}$, where $\eta_1 \leftrightarrow (\nu_1 > 0)$ and $\eta_2 \leftrightarrow (\nu_2 = 5)$. This is a valid conditioning set as none of the events has a zero probability of occurring, and we can safely perform the division in Equation 5.1.*

5.1. Conditioning on Zero-Probability Events

A prominent class of conditional queries, which are not captured by Definition 5.1, are so-called zero probability conditional queries. For such queries the probability of the observed event happening is actually zero but the event is still possible. Using Equation 5.1 does not work anymore as a division by zero would now occur.

Example 5.4 (Zero-Probability Conditioning Set)**.** *Assume that we have a random variable ν distributed according to a normal distribution and that we have the conditioning set $\mathcal{E} = \{\eta = 1\}$, with $\eta \leftrightarrow (\nu = 20)$. In other words, we condition the query on the observation that the random variable ν takes the value 20 – for instance in a distance measuring experiment. This is problematic as the probability of any specific value for a random variable with uncountably many outcomes is in fact zero and applying Equation 5.1 leads to a division-by-zero. Consequently, an ill-defined conditional probability arises.*

In order to sidestep divisions by zero when conditioning on zero-probability (but possible) events, we modify Definition 5.1. Analogously to Nitti et al. [2016], we follow the approach taken in [Kadane, 2011].

Definition 5.5 (Conditional Probability with Zero-Probability Events)**.** *Let ν be a continuous random variable in the DC-ProbLog program $\mathcal{P}$ with ground atoms $\mathcal{A}$. Furthermore, let us assume that the evidence consists of $\mathcal{E} = \{\eta_0 = \top\}$ with $\eta_0 \leftrightarrow (\nu = w)$ and $w \in \Omega_\nu$. The conditional probability of an atom of interest $\mu \in \mathcal{A}$ is now defined as:*

$$P_{\mathcal{P}}(\mu = \top \mid \eta_0 = \top) = \lim_{\Delta w \to 0} \frac{P_{\mathcal{P}}(\mu = \top, \nu \in [w - \Delta w/2, w + \Delta w/2])}{P_{\mathcal{P}}(\nu \in [w - \Delta w/2, w + \Delta w/2])} \tag{5.2}$$

To write this limit more compactly, we introduce an infinitesimally small constant δw and two new comparison atoms $\eta_1 \leftrightarrow (w - {}^{\delta w}/_2 \leq \nu)$ and $\eta_2 \leftrightarrow (\nu \leq w + {}^{\delta w}/_2)$ that together encode the limit interval. Using these, we rewrite Equation 5.2 as

$$P_{\mathcal{P}}(\mu = \top \mid \eta_0 = \top) = \frac{P_{\mathcal{P}}(\mu = \top, \eta_1 = \top, \eta_2 = \top)}{P_{\mathcal{P}}(\eta_1 = \top, \eta_2 = \top)} \tag{5.3}$$

Applying the definition recursively, allows us to have multiple zero probability conditioning events. More specifically, let us assume an additional continuous random variable ν' that takes the value w' for which we define: $\eta'_1 \leftrightarrow (w' - {}^{\delta w'}/_2 \leq \nu')$ and $\eta'_2 \leftrightarrow (\nu' \leq w' + {}^{\delta w'}/_2)$. This then leads to the following conditional probability:

$$\begin{aligned} P_{\mathcal{P}}(\mu = \top \mid \nu = w, \nu' = w') &= \frac{P_{\mathcal{P}}(\mu = \top, \eta_1 = \top, \eta_2 = \top \mid \nu' = w')}{P_{\mathcal{P}}(\eta_1 = \top, \eta_2 = \top \mid \nu' = w')} \\ &= \frac{\frac{P_{\mathcal{P}}(\mu=\top,\eta_1=\top,\eta_2=\top,\eta'_1=\top,\eta'_2=\top)}{\cancel{P_{\mathcal{P}}(\eta'_1=\top,\eta'_2=\top)}}}{\frac{P_{\mathcal{P}}(\eta_1=\top,\eta_2=\top,\eta'_1=\top,\eta'_2=\top)}{\cancel{P_{\mathcal{P}}(\eta'_1=\top,\eta'_2=\top)}}} \\ &= \frac{P_{\mathcal{P}}(\mu = \top, \eta_1 = \top, \eta_2 = \top, \eta'_1 = \top, \eta'_2 = \top)}{P_{\mathcal{P}}(\eta_1 = \top, \eta_2 = \top, \eta'_1 = \top, \eta'_2 = \top)} \end{aligned} \tag{5.4}$$

Here we first applied the definition of the conditional probability for the observation of the random variable ν and then for the observation of the random variable ν'. Finally, we simplified the expression.

Proposition 5.6. *The conditional probability as defined in Definition 5.5 exists.*

Proof. See [Nitti et al., 2016, Equation 6]. ◻

In order to express zero-probability events in DC-ProbLog we add a new built-in comparison predicate to the finite set of comparison predicates $\Pi = \{$<, >, =<, >=, =:=, =\=$\}$ (cf. Definition 3.1).

Definition 5.7 (Delta Interval Comparison)**.** *For a random variable* `v` *and a rational number* `w`*, we define* `delta_interval(v,w)` *(with* `delta_interval`/2 $\in \Pi$*) as follows. If* `v` *has a countable sample space, then* `delta_interval(v,w)` *is equivalent to* `v=:=w`. *Otherwise,* `delta_interval(v,w)` *is equivalent to the conjunction of the two comparison atoms* `w-δw=<v` *and* `v=<w+δw`*, where* `δw` *is an infinitesimally small number.*

The delta interval predicate lets us express conditional probabilities with zero probability conditioning events as defined in Definition 5.5.

Zero probability conditioning events are often abbreviated as $P_{\mathcal{P}}(\mu = \top \mid \nu = w)$. This can be confusing as it does not convey the intent of conditioning on an infinitesimally small interval. To this end, we introduce the symbol '$\doteq$' (equal sign with a dot on top). We use this symbol to explicitly point out an infinitesimally small conditioning set. For instance, we abbreviate the limit

$$\lim_{\Delta w \to 0} \frac{P_{\mathcal{P}}(\mu = \top, \nu \in [w - {}^{\Delta w}/_2, w + {}^{\Delta w}/_2])}{P_{\mathcal{P}}(\nu \in [w - {}^{\Delta w}/_2, w + {}^{\Delta w}/_2])}$$

in Definition 5.5 as:

$$P_{\mathcal{P}}(\mu = \top \mid \nu \doteq w) \tag{5.5}$$

More concretely, if we measure the height h of a person to be $180cm$ we denote this by $h \doteq 180$. This means that we measured the height of the person to be in an infinitesimally small interval around $180cm$. Note that the $\doteq$ sign has slightly different semantics for random variables with a countable support. For discrete random variables the $\doteq$ is equivalent to the *equal* sign.

Example 5.8. *Assume that we have a random variable ν distributed according to a normal distribution and that we have the evidence set $\mathcal{E} = \{\eta = \top\}$, with $\eta \leftrightarrow (\nu \doteq 20)$. This is a valid conditional probability defined through Definition 5.5.*

Example 5.9. *Assume that we have a random variable ν distributed according to a normal distribution and that we have the conditioning set $\mathcal{E} = \{\eta = \top, \eta' = \top\}$, with $\eta_1 \leftrightarrow (\nu \doteq 20)$ and $\eta' \leftrightarrow (\nu \doteq 30)$. This does not encode a conditional probability as the conditioning event is not a possible event: one and the same random variable cannot be observed to have two different outcomes.*

The notation used to condition on zero probability events (even when using '$\doteq$') hides away the limiting process that is used to define the conditional probability. This can lead to situations where seemingly equivalent conditional probabilities have diametrically opposed meanings.

Example 5.10. *Let us consider the conditioning set $\mathcal{E} = \{\eta = \top, \eta' = \top\}$, with $\eta \leftrightarrow (\nu \leq 20)$ and $\eta' \leftrightarrow (20 \leq \nu)$, which we use again to condition a continuous random variable ν. In contrast to Example 5.8, where we directly observed $\nu \doteq 20$, here, Definition 5.1 applies, which states that the conditional probability is undefined as $P(\nu \leq 20, 20 \leq \nu) = 0$.*

5.2. Discussion on the Well-Definedness of a Query

The probability of an unconditional query to a valid DC-ProbLog program is always well-defined, as it is simply a marginal of the distribution represented by the program. This stands in stark contrast to conditional probabilities: an obvious issue are divisions by zero occurring when the conditioning event does not belong to the set of possible outcomes of the conditioned random variable. Similarly to Wu et al. [2018] we will assume for the remainder of the paper that conditioning events are always possible events, i.e. events that have a non-zero probability but possibly an infinitesimally small probability of occurring. This allows us to bypass potential issues caused by zero-divisions.[2]

Even when discarding impossible conditioning events, conditioning a probabilistic event on a zero probability (but possible) event remains inherently ambiguous [Jaynes,

[2]In general, deciding whether a conditioning event is possible or not is undecidable. This follows from the undecidability of general logic programs under the well-founded semantics [Cherchago et al., 2007]. A similar discussion is also presented in the thesis of Brian Milch [Milch, 2006, Proposition 4.8] for the BLOG language, which also discusses decidable language fragments [Milch, 2006, Section 4.5].

2003] and might lead to the Borel-Kolmogorov paradox. Problems arise when the limiting process used to define the conditional probability with zero probability events (cf. Definition 5.5) does not produce a unique limit. For instance, a conditional probability $P(\mu = \top \mid 2\nu \doteq \nu')$, where ν and ν' are two random variables, depends on the parametrization used. We refer the reader to [Shan and Ramsey, 2017] and [Jacobs, 2021] for a more detailed discussion on ambiguities arising with zero probability conditioning events in the context of probabilistic programming. We will sidestep such ambiguities completely by limiting observations of zero probability events to direct comparisons between random variables and numbers. This makes also sense from an epistemological perspective: we interpret a conditioning event as the outcome of an experiment, which produces a number, for instance the reading of a tape measure.

5.3. Conditional Probabilities by Example

Example 5.11. *The following ProbLog program models the conditions under which machines work. There are two machines (Line 1), and three (binary) random terms, which we interpret as random variables as the bodies of the probabilistic facts are empty. The random variables are: the outside temperature (Line 3) and whether the cooling of each machine works (Lines 4 and 5). Each machine works if its cooling works or if the temperature is low (Lines 7 and 8).*

```
1 machine(1). machine(2).
2
3 0.8::temperature(low).
4 0.99::cooling(1).
5 0.95::cooling(2).
6
7 works(N):- machine(N), cooling(N).
8 works(N):- machine(N), temperature(low).
```

We can query this program for the probability of `works(1)` *given that we have as evidence that* `works(2)` *is true:*

$$P(\texttt{works(1)}=1 \mid \texttt{works(2)}=1) \approx 0.998$$

Example 5.12. *In the previous example there are only Boolean random variables (encoded as probabilistic facts) and the DC-ProbLog program is equivalent to an identical ProbLog program. An advantage of DC-ProbLog is that we can now use an almost identical program to model the temperature as a continuous random variable.*

```
machine(1). machine(2).

temperature ~ normal(20,5).
0.99::cooling(1).
0.95::cooling(2).

works(N):- machine(N), cooling(N).
works(N):- machine(N), temperature<25.0.
```

We can again ask for the probability of `works(1)` *given that we have as evidence that* `works(2)` *is true but now the program also involves a continuous random variable:*

$$P(\texttt{works(1)}{=}\top \mid \texttt{works(2)}{=}\top) \approx 0.998$$

In the two previous examples we were interested in a conditional probability where the conditioning event has a non-zero probability of occurring. However, DC-ProbLog programs can also encode conditional probabilities where the conditioning event has a zero probability of happening, while still being possible.

Example 5.13. *We model the size of a ball as a mixture of different beta distributions, depending on whether the ball is made out of wood or metal (Line 1). We would now like to know the probability of the ball being made out of wood given that we have a measurement of the size of the ball.*

```
1 3/10::material(wood);7/10::material(metal).
2
3 size~beta(2,3):- material(metal).
4 size~beta(4,2):- material(wood).
```

Assume that we measure the size of the ball and we find that it is 0.4*cm, which means that we have a measurement (or observation) infinitesimally close to* 0.4*. Using the '≐' notation, we write this conditional probability as:*

$$P\Big(\texttt{material(wood)}{=}\top \mid (\texttt{size}{\doteq}\texttt{4/10}){=}\top\Big) \tag{5.6}$$

The *Indian GPA problem* was initially proposed by Stuart Russell as an example problem to showcase the intricacies of mixed random variables. Below we express the Indian GPA problem in DC-ProbLog.

Example 5.14. *The Indian GPA problem models US-American and Indian students and their GPAs. Both receive scores on the continuous domain, namely from 0 to 4 (American) and from 0 to 10 (Indian), cf. Line 9 and 13. With non-zero probabilities both student groups can also obtain marks at the extremes of the respective scales (Lines 10, 11, 14, 15).*

```
1  1/4::american;3/4::indian.
2
3  19/20::isdensity(a).
4  99/100::isdensity(i).
5
6  17/20::perfect_gpa(a).
7  1/10::perfect_gpa(i).
8
9  gpa(a)~uniform(0,4):- isdensity(a).
10 gpa(a)~delta(4.0):- not isdensity(a), perfect_gpa(a).
11 gpa(a)~delta(0.0):- not isdensity(a), not perfect_gpa(a).
12
```

```
gpa(i)~uniform(0,10):- isdensity(i).
gpa(i)~delta(10.0):- not isdensity(i), perfect_gpa(i).
gpa(i)~delta(0.0):- not isdensity(i), not perfect_gpa(i).

gpa(student)~delta(gpa(a)):- american.
gpa(student)~delta(gpa(i)):- indian.
```

Note that in order to write the probability distribution of `gpa(a)` *and* `gpa(i)` *we used uniform and Dirac delta distributions. This allowed us to distribute the random variables* `gpa(a)` *and* `gpa(i)` *according to a discrete-continuous mixture distribution. We then observe that a student has a GPA of* 4 *and we would like to know the probability of this student being American or Indian.*

$$P\Big(\texttt{american}{=}\top \mid (\texttt{gpa(student)} \doteq 4) = \top\Big) = 1$$
$$P\Big(\texttt{indian}{=}\top \mid (\texttt{gpa(student)} \doteq 4) = \top\Big) = 0$$

6. Inference via Computing Expectations of Labeled Logic Formulas

In the previous sections we have delineated the semantics of DC-ProbLog programs and described the PROB task that defines conditional probability queries on DC-ProbLog programs. The obvious next step is to actually perform the inference. We will follow an approach often found in implementations of PLP languages in the discrete domain: reducing inference in probabilistic programs to performing inference on labeled Boolean formulas that encode relevant parts of the logic program. Contrary to languages in the discrete domain that follow this approach [Fierens et al., 2015, Riguzzi and Swift, 2011], we will face the additional complication of handling random variables with infinite sample spaces. We refer the reader to [Riguzzi, 2018, Section 5] for a broader overview of this approach.

Specifically, we are going to define a reduction from DC-ProbLog inference to the task of computing the expected label of a propositional formula. The formula is a propositional encoding of the relevant part of the logic program (relevant with respect to a query), where atoms become propositional variables, and the labels of the basic facts in the distribution database are derived from the probabilistic part of the program. At a high level, we extend ProbLog's inference algorithm such that Boolean comparison atoms over (potentially correlated) random variables are correctly being kept track of. The major complication, with regard to ProbLog and other systems such as PITA [Riguzzi and Swift, 2011], is the presence of context-dependent random variables, which are denoted by the same ground random term. For instance, the random term `size` in the program in Example 5.13 denotes two different random variables but is being referred to by one and the same term in the program.

Inference algorithms for PLP languages often consider only a fragment of the language for which the semantics have been defined. A common restriction for inference algorithms is to only consider range-restricted programs[3]. Furthermore, we consider,

[3]We call a DC-ProbLog program range-restricted if it holds that for every statement all logic variables

without loss of generality only AD-free programs, cf. Definition 4.15, as annotated disjunctions or probabilistic facts can be eliminated up front by means of *local* transformations that solely affect the annotated disjunctions (or probabilistic facts).[4]

The high level steps for converting a DC-ProbLog program to a labeled propositional formula closely follow the corresponding conversion for ProbLog programs provided by Fierens et al. [2015, Section 5], i.e., given a DC-ProbLog program $\mathcal{P}$, evidence $\mathcal{E} = e$ and a set of query atoms $\mathcal{Q}$, the conversion algorithm performs the following steps:

1. Determine the relevant ground program $\mathcal{P}_g$ with respect to the atoms in $\mathcal{Q} \cup \mathcal{E}$ and obtain the corresponing DF-PLP program.

2. Convert $\mathcal{P}_g$ to an equivalent propositional formula ϕ_g and $\mathcal{E} = e$ to a propositional conjunction ϕ_e.

3. Define the labeling function for all atoms in ϕ_g.

Step 1 exploits the fact that ground clauses that have no influence on the truth values of query or evidence atoms are irrelevant for inference and can thus be omitted from the ground program. Step 2 performs the conversion from logic program semantics to propositional logic, generating a formula that encodes *all* models of the relevant ground program as well as a formula that serves to assert the evidence by conjoining both formulas. Step 3 completes the conversion by defining the labeling function. In the following, we discuss the three steps in more detail and prove correctness of our approach (cf. Theorem 6.10).

6.1. The Relevant Ground Program

The first step in the conversion of a non-ground DC-ProbLog program to a labeled Boolean formula consists of grounding the program with respect to a query set $\mathcal{Q}$ and the evidence $\mathcal{E} = e$. For each ground atom in $\mathcal{Q}$ and $\mathcal{E}$ we construct its dependency set. That is, we collect the set of ground atoms and ground rules that occur in any of the proofs of an atom in $\mathcal{Q} \cup \mathcal{E}$. The union of all dependency sets for all the ground atoms in $\mathcal{Q} \cup \mathcal{E}$ is the dependency set of the DC-ProbLog with respect to the sets $\mathcal{Q}$ and $\mathcal{E}$. This dependency set, consisting of ground rules and ground atoms, is called the relevant ground program (with respect to a set of queries and evidence).

Example 6.1. *Consider the non-ground (AD-free) DC-ProbLog program below.*

```
rv_hot ~ flip(0.2).
hot:- rv_hot=:=1.
```

occurring in the head also occur in the body. This guarantees that all terms will become ground during backward chaining. Note that range-restrictedness implies that all facts (including probabilistic and distributional ones) are ground.

[4]For non-ground ADs, we adapt Definition 4.14 to include all logical variables as arguments of the new random variable. As this introduces non-ground distributional facts, which are not range-restricted, we also move the comparison atom to the end of the rule bodies of the AD encoding to ensure those local random variables are ground when reached in backward chaining.

```
rv_cool(1) ~ flip(0.99).
cool(1):- rv_cool(1)=:=1.

temp(1) ~ normal(27,5):- hot.
temp(1) ~ normal(20,5):- not hot.

works(N):- cool(N).
works(N):- temp(N)<25.0.
```

If we ground it with respect to the query `works(1)` *and subsequently apply the rewrite rules from Section 4.2.2 we obtain:*

```
rv_hot ~ flip(0.2).
hot:- rv_hot=:=1.
rv_cool(1) ~ flip(0.99).
cool(1):- rv_cool(1)=:=1.

temp(hot) ~ normal(27,5).
temp(not_hot) ~ normal(20,5).

works(1):- cool(1).
works(1):- hot, temp(hot)<25.0,
works(1):- not hot, temp(not_hot)<25.0.
```

A possible way, as hinted at in Example 6.1 of obtaining a ground DF-PLP program from a non-ground DC-ProbLog program is to first ground out all the logical variables. Subsequently, one can apply Definition 4.14 to eliminate annotated disjunctions and probabilistic facts, Definition 4.14 and Definition 4.25 in order to obtain a DF-PLP program with no distributional clauses. A possible drawback of such a two-step approach (grounding logical variables followed by obtaining a DC-ProbLog program) is that it might introduce spurious atoms to the relevant ground program. A more elegant but also more challenging approach is to interleave the grounding of logical variables and distributional clause elimination. We leave this for future research.

Theorem 6.2 (Label Equivalence)**.** *Let $\mathcal{P}$ be a DC-ProbLog program and let $\mathcal{P}_g$ be the relevant ground program for $\mathcal{P}$ with respect to a query μ and the evidence $\mathcal{E} = e$ obtained by first grounding out logical variables and subsequently applying transformation rules from Section 4. The programs $\mathcal{P}$ and $\mathcal{P}_g$ specify the same probability:*

$$P_{\mathcal{P}}(\mu = \top \mid \mathcal{E} = e) = P_{\mathcal{P}_g}(\mu = \top \mid \mathcal{E} = e) \tag{6.1}$$

Proof. See Appendix F.1. □

6.2. The Boolean Formula for the Relevant Ground Program

Converting a ground logic program, i.e. a set of ground rules, into an equivalent Boolean formula is a purely logical problem and well-studied in the non-probabilistic logic programming literature. We refer the reader to Janhunen [2004] for an account of the transformation to Boolean formula in the non-probabilistic setting and to Mantadelis and Janssens [2010] and Fierens et al. [2015] in the probabilistic setting, including correctness proofs. We will only illustrate the most basic case with an example here.

Example 6.3 (Mapping DC-ProbLog to Boolean Formula)**.** *Consider the ground program in Example 6.1. To highlight the move from logic programming to propositional logic, we introduce for every atom* `a` *in the program a corresponding propositional variable* $\phi_{\texttt{a}}$*. As the program does not contain cycles, we can use Clark's completion for the transformation, i.e., a derived atom is true if and only if the disjunction of the bodies of its defining rules is true. The propositional formula* ϕ_g *corresponding to the program is then the conjunction of the following three formulas:*

$$
\begin{aligned}
\phi_{\texttt{works(1)}} &\leftrightarrow \left(\phi_{\texttt{cool(1)}} \vee \phi_{\texttt{hot}} \wedge \phi_{\texttt{temp(hot)<25.0}} \vee \neg\phi_{\texttt{hot}} \wedge \phi_{\texttt{temp(not\_hot)<25.0}}\right)\\
\phi_{\texttt{cool(1)}} &\leftrightarrow \phi_{\texttt{rv\_cool(1)=:=1}}\\
\phi_{\texttt{hot}} &\leftrightarrow \phi_{\texttt{rv\_hot=:=1}}
\end{aligned}
$$

Note that the formula obtained by converting the relevant ground program still admits *any* model of that program, including ones that are inconsistent with the evidence. In order to use that formula to compute conditional probabilities, we still need to assert the evidence into the formula by conjoining the corresponding propositional literals. The following theorem then directly applies to our case as well.

Theorem 6.4 (Model Equivalence [Fierens et al., 2015] (Theorem 2, part 1))**.** *Let* $\mathcal{P}_g$ *be the relevant ground program for a DC-ProbLog program* $\mathcal{P}$ *with respect to query set* $\mathcal{Q}$ *and evidence* $\mathcal{E} = e$*. Let* $MOD_{\mathcal{E}=e}(\mathcal{P}_g)$ *be those models in* $MOD(\mathcal{P}_g)$ *that are consistent with the evidence. Let* ϕ_g *denote the propositional formula derived from* $\mathcal{P}_g$*, and set* $\phi \leftrightarrow \phi_g \wedge \phi_e$*, where* ϕ_e *is the conjunction of literals that corresponds to the observed truth values of the atoms in* $\mathcal{E}$*. We then have* **model equivalence***, i.e.,*

$$
MOD_{\mathcal{E}=e}(\mathcal{P}_g) = ENUM(\phi) \tag{6.2}
$$

where $ENUM(\phi)$ *denotes the set of models of* ϕ*.*

6.3. Obtaining a Labeled Boolean Formula

In contrast to a ProbLog program, a DC-ProbLog program does not explicitly provide independent probability labels for the basic facts in the distribution semantics, and we thus need to suitably adapt the last step of the conversion. We will first define the labeling function on propositional atoms and will then show that the probability of the label of a propositional formula is the same as the probability of the relevant ground program under the distribution semantics from Section 3. We call this *label equivalence* and prove it in Theorem 6.9.

Definition 6.5 (Label of Literal)**.** *The label* $\alpha(\phi_\rho)$ *of a propositional atom* ϕ_ρ *(or its negation) is given by:*

$$
\alpha(\phi_\rho) = \begin{cases} [\![c(vars(\rho))]\!], & \text{if } \rho \text{ is a comparison atom} \\ 1, & \text{otherwise} \end{cases} \tag{6.3}
$$

and for the negated atom:

$$\alpha(\neg\phi_\rho) = \begin{cases} [\![\neg c(vars(\rho))]\!], & \text{if } \rho \text{ is a comparison atom} \\ 1, & \text{otherwise} \end{cases} \tag{6.4}$$

We use Iverson brackets $[\![\cdot]\!]$ [Iverson, 1962] to denote an indicator function. Furthermore, $vars(\rho)$ denotes the random variables that are present in the arguments of the atom ρ and $c(\cdot)$ encodes the constraint given by ρ.

Example 6.6 (Labeling function)**.** *Continuing Example 6.3, we obtain, inter alia, the following labels:*

$$\begin{aligned}
&\alpha(\phi_{\texttt{rv_hot=:=1}}) = [\![rv_hot = 1]\!] \\
&\alpha(\neg\phi_{\texttt{rv_hot=:=1}}) = [\![\neg(rv_hot = 1)]\!] = [\![rv_hot = 0]\!] \\
&\alpha(\phi_{\texttt{hot}}) = 1 \\
&\alpha(\neg\phi_{\texttt{hot}}) = 1
\end{aligned}$$

Definition 6.7 (Label of Boolean Formula)**.** *Let ϕ be a Boolean formula and $\alpha(\cdot)$ the labeling function for the variables in ϕ as given by Definition 6.5. We define the label of ϕ as*

$$\alpha(\phi) = \sum_{\varphi \in ENUM(\phi)} \prod_{\ell \in \varphi} \alpha(\ell)$$

i.e. as the sum of the labels of all its models, which are in turn defined as the product of the labels of their literals.

Example 6.8 (Labeled Boolean Formula)**.** *The label of the conjunction*

$$\neg\phi_{\texttt{hot}} \land \neg\phi_{\texttt{rv\_hot=:=1}} \land \phi_{\texttt{temp(not\_hot)<25.0}} \land \neg\phi_{\texttt{cool(1)}} \land \neg\phi_{\texttt{rv\_cool(1)=:=1}} \land \phi_{\texttt{works(1)}}$$

which describes one model of the example formula, is computed as follows:

$$\begin{aligned}
&\alpha(\neg\phi_{\texttt{hot}} \land \neg\phi_{\texttt{rv_hot=:=1}} \land \phi_{\texttt{temp(not_hot)<25.0}} \\
&\quad \land \neg\phi_{\texttt{cool(1)}} \land \neg\phi_{\texttt{rv_cool(1)=:=1}} \land \phi_{\texttt{works(1)}}) \\
&= \alpha(\neg\phi_{\texttt{hot}}) \times \alpha(\neg\phi_{\texttt{rv_hot=:=1}}) \times \alpha(\phi_{\texttt{temp(not_hot)<25.0}}) \\
&\quad \times \alpha(\neg\phi_{\texttt{cool(1)}}) \times \alpha(\neg\phi_{\texttt{rv_cool(1)=:=1}} \times \alpha(\phi_{\texttt{works(1)}})) \\
&= 1 \times [\![rv_hot = 0]\!] \times [\![temp(not_hot) < 25]\!] \times 1 \times [\![rv_cool(1) = 0]\!] \times 1 \\
&= [\![rv_hot = 0]\!] \times [\![temp(not_hot) < 25]\!] \times [\![rv_cool(1) = 0]\!]
\end{aligned}$$

Theorem 6.9 (Label Equivalence)**.** *Let $\mathcal{P}_g$ be the relevant ground program for a DC-ProbLog program $\mathcal{P}$ with respect to a query μ and the evidence $\mathcal{E} = e$. Let ϕ_g denote the propositional formula derived from $\mathcal{P}_g$ and let α be the labeling function as defined in Definition 6.5. We then have* **label equivalence***, i.e.*

$$\forall \varphi \in ENUM(\phi_g) : \mathop{\mathbb{E}}_{\mathcal{V}\sim\mathcal{P}_g}[\alpha(\varphi)] = P_{\mathcal{P}_g}(\varphi) \tag{6.5}$$

In other words, for all models φ of ϕ_g, the expected value ($\mathbb{E}.[\cdot]$) of the label of φ is equal to the probability of φ according to the probability measure of relevant ground program $\mathcal{P}_g$.

Proof. See Appendix F.2. □

Theorem 6.9 states that we can reduce inference in hybrid probabilistic logic programs to computing the expected value of labeled Boolean formulas, as summarized in the following theorem.

Theorem 6.10. *Given a DC-ProbLog program $\mathcal{P}$, a set $\mathcal{Q}$ of queries, and evidence $\mathcal{E} = e$, for every $\mu \in \mathcal{Q}$, we obtain the conditional probability of $\mu = q$ ($q \in \{\bot, \top\}$) given $\mathcal{E} = e$ as*

$$P(\mu = q \mid \mathcal{E} = e) = \frac{\mathbb{E}_{vars(\phi)\sim\mathcal{P}_g}[\alpha(\phi \wedge \phi_q)]}{\mathbb{E}_{vars(\phi)\sim\mathcal{P}_g}[\alpha(\phi)]}$$

where ϕ is the formula encoding the relevant ground program $\mathcal{P}_g$ with the evidence asserted (cf. Theorem 6.4), and ϕ_q the propositional atom for μ.

Proof. This directly follows from model and label equivalence together with the definition of conditional probabilities. □

We have shown that the probability of a query to a DC-ProbLog program can be expressed as the expected label of a propositional logic formula.

7. Computing Expected Labels via Algebraic Model Counting

In this section we will adapt the approach taken by Zuidberg Dos Martires et al. [2019b], dubbed *Sampo* to compute the expected value of labeled propositional Boolean formulas. The method approximates intractable integrals that appear when computing expected labels using Monte Carlo estimation. The main difference between Sampo and our approach, which we dub *infinitesimal algebraic likelihood weighting* (IALW) is that IALW can also handle infinitesimally small intervals, which arise when conditioning on zero probability events.

7.1. Monte Carlo Estimate of Conditional Query

In Definition 5.1 we defined the conditional probability as:

$$P_{\mathcal{P}}(\mu = \top \mid \mathcal{E} = e) = \frac{P_{\mathcal{P}}(\mu = \top, \mathcal{E} = e)}{P_{\mathcal{P}}(\mathcal{E} = e)} \tag{7.1}$$

and we also saw in Definition 5.5 that using infinitesimal intervals allows us to consider zero probability events, as well. Computing the probabilities in the numerator and denominator in the equation above is, in general, computationally hard. We resolve this using a Monte Carlo approximation.

Proposition 7.1 (Monte Carlo Approximation of a Conditional Query)**.** *Let the set*

$$\mathcal{S} = \left\{ \left(s_1^{(1)}, \dots, s_M^{(1)}\right), \dots, \left(s_1^{(|\mathcal{S}|)}, \dots, s_M^{(|\mathcal{S}|)}\right) \right\} \tag{7.2}$$

denote $|\mathcal{S}|$ i.i.d. samples for each random variable in $\mathcal{P}_g$. A conditional probability query to a DC-ProbLog program $\mathcal{P}$ can be approximated as:

$$P_{\mathcal{P}}(\mu = q \mid \mathcal{E} = e) \approx \frac{\sum_{i=1}^{|\mathcal{S}|} \sum_{\varphi \in ENUM(\phi \land \phi_q)} \alpha^{(i)}(\varphi)}{\sum_{i=1}^{|\mathcal{S}|} \sum_{\varphi \in ENUM(\phi)} \alpha^{(i)}(\varphi)}, \qquad N < \infty \tag{7.3}$$

The index (i) on $\alpha^{(i)}(\varphi)$ indicates that the label of φ is evaluated at the i-th ordered set of samples $\left(s_1^{(i)}, \ldots, s_M^{(i)}\right)$.

Proof. See Appendix F.3. □

In the limit $|\mathcal{S}| \to \infty$ this sampling approximation scheme is perfectly valid. However, in practice, with only limited resources available, such a rejection sampling strategy will perform poorly (in the best case) or even give completely erroneous results. After all, the probability of sampling a value from the prior distribution that falls exactly into an infinitesimally small interval given in the evidence tends to zero. To make the computation of a conditional probability, using Monte Carlo estimates, feasible, we are going to introduce *infinitesimal algebraic likelihood weighting*. But first, we will need to introduce the concept of infinitesimal numbers.

7.2. Infinitesimal Numbers

Remember that infinitesimal intervals arise in zero probability conditioning events and describe an infinitesimally small interval around a specific observed value, e.g. $\nu \in [w - {}^{\Delta w}/_2, w + {}^{\Delta w}/_2]$ for a continuous random variable ν that was observed to take the value w (cf. Definition 5.5). We will describe these infinitesimally small intervals using so-called *infinitesimal numbers*, which were first introduced by Nitti et al. [2016] and further formalized in Wu et al. [2018], [Zuidberg Dos Martires, 2020] and [Jacobs, 2021]. The latter work also coined the term *'infinitesimal number'*.

Definition 7.2 (Infinitesimal Numbers)**.** *An infinitesimal number is a pair $(r, n) \in \mathbb{R}\times\mathbb{Z}$, also written as $r\epsilon^n$, and which corresponds to a real number when $n = 0$. We denote the set of all infinitesimal numbers by $\mathbb{I}$.*

Definition 7.3 (Operations in $\mathbb{I}$)**.** *Let (r, n) and (t, m) be two numbers in $\mathbb{I}$. We define the addition and multiplication as binary operators:*

$$(r, n) \oplus (t, m) := \begin{cases} (r + t, n) & \text{if } n = m \\ (r, n) & \text{if } n < m \\ (t, m) & \text{if } n > m \end{cases} \tag{7.4}$$

$$(r, n) \otimes (t, m) := (r \times t, n + m) \tag{7.5}$$

The operations $+$ and $\times$ on the right hand side denote the usual addition and multiplication operations for real and integer numbers.

Definition 7.4 (Neutral Elements)**.** *The neutral elements of the addition and multiplications in $\mathbb{I}$ are, respectively, defined as:*

$$e^{\oplus} := (0, 0) \qquad\qquad e^{\otimes} := (1, 0) \tag{7.6}$$

Probabilistic inference and generalization thereof can often be cast as performing computations using commutative semirings [Kimmig et al., 2017]. We will follow a similar strategy.

Definition 7.5. *A* **commutative semiring** *is an algebraic structure* $(\mathcal{A}, \oplus, \otimes, e^{\oplus}, e^{\otimes})$ *equipping a set of elements* $\mathcal{A}$ *with addition and multiplication such that*

1. *addition* $\oplus$ *and multiplication* $\otimes$ *are binary operations* $\mathcal{A} \times \mathcal{A} \to \mathcal{A}$

2. *addition* $\oplus$ *and multiplication* $\otimes$ *are associative and commutative binary operations over the set* $\mathcal{A}$

3. $\otimes$ *distributes over* $\oplus$

4. $e^{\oplus} \in \mathcal{A}$ *is the neutral element of* $\oplus$

5. $e^{\otimes} \in \mathcal{A}$ *is the neutral element of* $\otimes$

6. $e^{\oplus} \in \mathcal{A}$ *is an annihilator for* $\otimes$

Lemma 7.6. *The structure* $(\mathbb{I}, \oplus, \otimes, e^{\oplus}, e^{\otimes})$ *is a commutative semiring.*

Proof. This follows trivially from the operations defined in Definition 7.3 and the neutral elements in Definition 7.4. □

We will also need to perform subtractions and divisions in $\mathbb{I}$, for which we first need to define inverse elements.

Definition 7.7 (Inverse Elements)**.** *Let* (r, n) *be a number in* $\mathbb{I}$*. We define its inverse with respect to the addition* $-(r, n)$*, also called negation, as:*

$$-(r, n) \coloneqq (-r, n) \tag{7.7}$$

Moreover, we define its inverse with respect to the multiplication $(r, n)^{-1}$*, also called the reciprocal, as:*

$$(r, n)^{-1} \coloneqq \begin{cases} (r^{-1}, -n) & \text{if } r \neq 0 \\ \text{undefined} & \text{if } r = 0 \end{cases} \tag{7.8}$$

Definition 7.8 (Subtraction and Division in $\mathbb{I}$)**.** *Let* (r, n) *and* (s, m) *be two numbers in* $\mathbb{I}$*. We define the subtraction and division as:*

$$(r, n) \ominus (t, m) \coloneqq (r, n) \oplus (-(t, m)) = (r, n) \oplus (-t, m) \tag{7.9}$$

$$(r, n) \oslash (s, m) \coloneqq \begin{cases} (r, n) \otimes (t, m)^{-1} = (r, n) \otimes (t^{-1}, -m) & \text{if } t \neq 0 \\ \text{undefined} & \text{if } t = 0 \end{cases} \tag{7.10}$$

The idea behind IALW is that we do not sample random variables that fall within an infinitesimal small interval, encoded as a delta interval (cf. Definition 5.7), but that we force, without sampling, the random variable to lie inside the infinitesimal interval. To this end, assume again that we have N i.i.d. samples for each random variable. That means that we have again a set of ordered sets of samples:

$$\mathcal{S} = \left\{ \left(s_1^{(1)}, \dots, s_M^{(1)} \right), \dots, \left(s_1^{(|\mathcal{S}|)}, \dots, s_M^{(|\mathcal{S}|)} \right) \right\} \tag{7.11}$$

This time the samples are drawn with the infinitesimal delta intervals taken into account. For example, assume we have a random variable v_1 distributed according to a normal distribution $\mathcal{N}(5, 2)$ and we have an atom `delta_interval(`v_1`,4)` in the propositional formula ϕ. Each sampled value of $s_1^{(i)}$ will then equal 4 ($1 \leq i \leq N$). Furthermore, when sampling, we sample the parents of a random variable prior to sampling the random variable itself. For instance, take the random variable $v_2 \sim \mathcal{N}(v_3 = w, 2)$, where v_3 is itself a random variable. We first sample v_3 and once we have a value for v_3 we plug that into the distribution for v_2, which we sample subsequently. In other words, we sample according to the ancestor relationship between the random variables. We call the ordered set of samples $\mathbf{s}^{(i)} \in \mathcal{S}$ an *ancestral sample*.

Definition 7.9 (IALW Label)**.** *Given is an ancestral sample $\mathbf{s}^{(i)} = (s_1^{(i)}, \dots, s_M^{(i)})$ for the random variables $\mathcal{V} = (v_1, \dots, v_M)$. We, furthermore, denote the probability distribution of a random variable v_k by δ_k and $\delta_k(\mathbf{s}^{(i)})$ evaluates the distribution for the i-th ancestral sample. The IALW label of a positive literal ℓ is an infinitesimal number given by:*

$$\alpha^{(i)}_{IALW}(\ell) = \begin{cases} (\delta_k(\mathbf{s}^{(i)}), 1), & \text{if } \ell \text{ is a } \texttt{delta\_interval} \text{ whose first argument} \\ & \text{is a continuous random variable} \\ (\ell(\mathbf{s}^{(i)}), 0), & \text{if } \ell \text{ is any comparison atom} \\ (1, 0), & \text{otherwise} \end{cases}$$

The expression $\ell(\mathbf{s}^{(i)})$ denotes the indicator function, which corresponds to the literal ℓ, being evaluated using the samples $\mathbf{s}^{(i)}$, and implies that $\ell(\mathbf{s}^{(i)}) \in \{0, 1\}$.

For the negated literals we have the following labeling function:

$$\alpha^{(i)}_{IALW}(\neg\ell) = \begin{cases} (1, 0), & \text{if } \ell \text{ is a } \texttt{delta\_interval} \text{ whose first argument} \\ & \text{is a continuous random variable} \\ (1 - \ell(\mathbf{x}^{(i)}), 0), & \text{if } \ell \text{ is any other comparison atom} \\ (1, 0), & \text{otherwise} \end{cases}$$

Intuitively speaking and in the context of probabilistic inference, the first part of an infinitesimal number accumulates (unnormalized) likelihood weights, while the second part counts the number of times we encounter a `delta_interval` atom. This counting happens with $\oplus$ operation of the infinitesimal numbers (Equation 7.4). The $\oplus$ operation tells us that for two infinitesimal number (r, n) and (t, m) with $n > m$, the event corresponding to the first of the two infinitesimal numbers is infinitely more probable to happen and that we drop the likelihood weight of the second infinitesimal number (Equation 7.4). In other words, an event with fewer `delta_interval`-atoms is infinitely more probable than an event with more such intervals.

Example 7.10 (IALW Label of `delta_interval` with Continuous Random Variable)**.** *Let us consider a random variable* `x`*, which is normally distributed:* $p(\mathtt{x}|\mu, \sigma) = {}^{1}/_{(\sigma\sqrt{2\pi})} \exp\left(-{}^{(\mathtt{x}-\mu)^2}/_{2\sigma^2}\right)$*), where* μ *and* $\sigma > 0$ *are real valued parameters that we can choose freely. The atom* `delta_interval(x,3)` *gets the label*

$$\left(\frac{1}{(\sigma\sqrt{2\pi})} \exp\left(-{}^{(3-\mu)^2}/_{2\sigma^2}\right), 1\right)$$

The first element of the infinitesimal number is the probability distribution evaluated at the observation, in this case 3*. As this is a zero probability event, the label also picks up a non-zero second element.*

The label of $\neg$`delta_interval(x,3)` *is* $(1, 0)$*. The intuition here being that the complement of an event with zero probability of happening will happen with probability* 1*. As the complement event is not a zero probability event the second element of the label is* 0 *instead of* 1*.*

Example 7.11 (IALW Label of `delta_interval` with Discrete Random Variable)**.** *Let us consider a discrete random variable* `k`*, which is Poisson distributed:*

$$p(\mathtt{k}|\lambda) = {}^{\lambda^{\mathtt{k}} e^{-\lambda}}/_{\mathtt{k}!}$$

where $\lambda > 0$ *is a real-valued parameter that we can freely choose.*

As a `delta_interval` *with a discrete random variable is equivalent to a* `=:=` *comparison (cf. Definition 5.7), we get for the label of the atom* `delta_interval(k,3)`*:* $([\![s_x^{(i)} = 3]\!], 0)$*, where* $s_{\mathtt{k}}^{(i)}$ *is the i-th sample for* `k`*.*

Definition 7.12 (Infinitesimal Algebraic Likelihood Weighting)**.** *Let* $\mathcal{S}$ *be a set of ancestral samples and let* $DI(\varphi)$ *denote the subset of literals in* φ *that are delta intervals. We then define IALW as expressing the expected value of the label of a propositional formula (given a set of ancestral samples) in terms of a fraction of two infinitesimal numbers:*

$$\left(\mathbb{E}\left[\sum_{\varphi \in ENUM(\phi)} \prod_{\ell \in \varphi} \alpha(\ell) \middle| \mathcal{S}\right], 0\right) \approx \frac{\bigoplus_{i=1}^{|\mathcal{S}|} \bigoplus_{\varphi \in ENUM(\phi)} \bigotimes_{\ell \in \varphi} \alpha_{IALW}^{(i)}(\ell)}{\bigoplus_{i=1}^{|\mathcal{S}|} \bigoplus_{\varphi \in ENUM(\phi)} \bigotimes_{\ell \in DI(\varphi)} \alpha_{IALW}^{(i)}(\ell)} \tag{7.12}$$

The left hand side expresses the expected value as an infinitesimal number.

Proposition 7.13 (Consistency of IALW). *Infinitesimal algebraic likelihood weighting is consistent, that is, the approximate equality in Equation 7.12 is almost surely an equality for* $|\mathcal{S}| \rightarrow \infty$.

Proof. See Appendix F.4. □

Likelihood weighting, the core idea behind IALW, is a well known technique for inference in Bayesian networks [Fung and Chang, 1990] and probabilistic programming [Milch et al., 2005b, Nitti et al., 2016], and falls within the broader class of self-normalized importance sampling [Kahn, 1950, Kloek and Van Dijk, 1978, Casella and Robert, 1998]. Just like IALW, the inference approaches proposed by Nitti et al. [2016], Wu et al. [2018], and Jacobs [2021] generalize the idea of likelihood weighting to the setting with infinitesimally small intervals. What sets IALW apart from these methods is its semiring formulation. The semiring formulation will allow us to seamlessly combine IALW with knowledge compilation [Darwiche and Marquis, 2002], a technique underlying state-of-the art probabilistic inference algorithms in the discrete setting. We examine this next.

Having proven the consistency of IALW, we can now express the probability of a conditional query to a DC-ProbLog program in terms of semiring operations for infinitesimal numbers $\mathbb{I}$.

Proposition 7.14. *A conditional probability query to a DC-ProbLog program* $\mathcal{P}$ *can be approximated as:*

$$P_{\mathcal{P}}(\mu = q | \mathcal{E} = e) \approx \frac{\bigoplus_{i=1}^{|\mathcal{S}|} \bigoplus_{\varphi \in ENUM(\phi \wedge \phi_q)} \bigotimes_{\ell \in \varphi} \alpha_{IALW}^{(i)}(\ell)}{\bigoplus_{i=1}^{|\mathcal{S}|} \bigoplus_{\varphi \in ENUM(\phi)} \bigotimes_{\ell \in \varphi} \alpha_{IALW}^{(i)}(\ell)} \tag{7.13}$$

Proof. See Appendix F.5. □

7.4. Infinitesimal Algebraic Likelihood Weighting via Knowledge Compilation

Inspecting Equation 7.13 we see that we have to evaluate expressions of the following form in order to compute the probability of a conditional query to a DC-ProbLog program.

$$\bigoplus_{i=1}^{|\mathcal{S}|} \underbrace{\bigoplus_{\omega \in ENUM(\varphi)} \bigotimes_{\ell \in \varphi} \alpha_{IALW}^{(i)}(\ell)}_{\text{=algebraic model count}} \tag{7.14}$$

In other words, we need to compute $|\mathcal{S}|$ times a sum over products – each time with a different ancestral sample. Such a sum over products is also called the algebraic model count of a formula ϕ [Kimmig et al., 2017]. Subsequently, we then add up the $|\mathcal{S}|$ results from the different algebraic model counts giving us the final answer.

Unfortunately, computing the algebraic model count is in general a computationally hard problem [Kimmig et al., 2017] – #P-hard to be precise [Valiant, 1979]. A popular technique to mitigate this hardness is to use a technique called knowledge compilation [Darwiche and Marquis, 2002], which splits up the computation into a hard step and a subsequent easy step. The idea is to take the propositional Boolean formula underlying an algebraic model counting problem (cf. ϕ in Equation 7.14) and compile it into a logically equivalent formula that allows for the tractable computation of algebraic model counts. The compilation constitutes the computationally hard part (#P-hard). Afterwards, the algebraic model count is performed on the compiled structure, also called *algebraic circuit* [Zuidberg Dos Martires et al., 2019a]. Intuitively speaking, knowledge compilation takes the sum of products and maps it to recursively nested sums and products. Effectively, finding a dynamic programming scheme [Bellman, 1957] to compute the initial sum of products.

Different circuit classes have been identified as valid knowledge compilation targets [Darwiche and Marquis, 2002] – all satisfying different properties. Computing the algebraic model count on an algebraic circuit belonging to a specific target class is only correct if the properties of the circuit class match the properties of the deployed semiring. The following three lemmas will help us determining which class of circuits we need to knowledge-compile our propositional formula ϕ into.

Lemma 7.15. *The operator $\oplus$ (c. Definition 7.3) is not idempotent. That is, it does not hold for every $a \in \mathbb{I}$ that $a \oplus a = a$.*

Lemma 7.16. *The pair $(\oplus, \alpha_{IALW})$ is not neutral. That is, it does not hold that $\alpha_{IALW}(\ell) \oplus \alpha_{IALW}(\neg\ell) = e^{\otimes}$ for arbitrary ℓ.*

Lemma 7.17. *The pair $(\otimes, \alpha_{IALW})$ is not consistency-preserving. That is, it does not hold that $\alpha_{IALW}(\ell) \otimes \alpha_{IALW}(\neg\ell) = e^{\oplus}$ for arbitrary ℓ.*

From [Kimmig et al., 2017, Theorem 2 and Theorem 7] and the three lemmas above, we can conclude that we need to compile our propositional logic formulas into so-called smooth, deterministic and decomposable negation normal form (sd-DNNF) formulas [Darwiche, 2001].[5]

Proposition 7.18 (ALW on d-DNNF)**.** *We are given the propositional formulas ϕ and ϕ_q and a set $\mathcal{S}$ of ancestral samples, we can use Algorithm 7.19 to compute the conditional probability $P_{\mathcal{P}}(\mu = q | \mathcal{E} = e)$.*

Proof. See Appendix F.6. □

Algorithm 7.19 takes as input a two propositional logic formulas ϕ and ϕ_q, and a set of ancestral samples. It then knowledge-compiles the formulas $\phi \wedge \phi_q$ and ϕ into circuits Γ_q and Γ. These circuits are then evaluated using Algorithm 7.20. The variables $ialw_q$ and $ialw$ hold infinitesimal numbers. The returned result is the ration of these two number, which corresponds to ration in Equation 7.13.

Algorithm 7.19

[5]Note that we only require smoothness over derived atoms (otherwise case in Definition 7.12), as for the other cases the neutral sum property holds. Certain encodings of logic programs eliminate derived atoms. For such encodings the smoothness property can be dropped [Vlasselaer et al., 2014]. A more detailed discussion on the smoothness requirement of circuits in a PLP context can be found in [Fierens et al., 2015, Appendix C].

Algorithm 7.19: Conditional Probability via IALW and KC

function ProbALW(ϕ, ϕ_q, $\mathcal{S}$)
  $\Gamma_q \leftarrow$ KC($\phi \land \phi_q$)
  $\Gamma \leftarrow$ KC(ϕ)
  $ialw_q \leftarrow$ IALW(Γ_q,$\mathcal{S}$)   // cf. Algorithm 7.20
  $ialw \leftarrow$ IALW(Γ,$\mathcal{S}$)   // cf. Algorithm 7.20
  return $ialw_q \oslash ialw$

Algorithm 7.20: Computing the IALW

function IALW(Γ, $\mathcal{S}$)
  $ialw \leftarrow (0{,}0)$
  for $\mathbf{s}^{(i)} \in \mathcal{S}$ **do**
    $ialw \leftarrow ialw \oplus$ Eval(Γ, $\mathbf{s}^{(i)}$)   // cf. Algorithm 7.21
  return $ialw$

Algorithm 7.20 compute the IALW given as input a circuit Γ and a set of ancestral samples. The loop evaluates the circuit (using Algorithm 7.21) for each ancestral sample $\mathbf{s}^{(i)}$ and accumulates the result, which is then returned once the loop terminates. The accumulation inside the loop corresponds to the $\bigoplus_{i=1}^{|\mathcal{S}|}$ summation in Equation 7.14. Algorithm 7.21 evaluates a circuit Γ for a single ancestral sample $\mathbf{s}^{(i)}$ and is a variation of the circuit evaluation algorithm presented by Kimmig et al. [2017].

Algorithm 7.21: Evaluating an sd-DNNF circuit Γ for labeling function $\alpha^{(i)}$ (Definition 7.9) and semiring operations $\oplus$ and $\otimes$ (Definition 7.3)

function Eval(Γ,$\mathbf{s}^{(i)}$)
  if *Γ is a literal node l* **then**
    return $\alpha^{(i)}(l)$
  else if *Γ is a disjunction $\bigvee_{j=1}^{m} \Gamma_j$* **then**
    return $\bigoplus_{j=1}^{m}$ Eval(Γ_j, $\mathbf{s}^{(i)}$)
  else   // Γ is a conjunction $\bigwedge_{j=1}^{m} \Gamma_j$
    return $\bigotimes_{i=j}^{m}$ Eval(Γ_j, $\mathbf{s}^{(i)}$)

Example 7.22 (IALW on Algebraic Circuit)**.** *Consider a version of the program in Example 5.13 where the annotated disjunction has been eliminated and been replaced with a binary random variable m and a* `flip` *distribution.*

```
m~flip(0.3).

```

```
size~beta(2,3):- m:=0.
size~beta(4,2):- m:=1.
```

We query the program for the conditional probability $P((\texttt{m:=1}) = \top \mid \texttt{size} \doteq 4/10)$. *Following the program transformations introduced in Section 6 and then compiling the labeled propositional formula, we obtain a circuit representation of the queried program. Evaluating this circuit yields the probability of the query. To be precise, we actually obtain two circuits, one representing the probability of relevant program with the evidence enforced and with additionally having the value of the query atom set. In Figure 7.4 we show the circuit where only the evidence is enforced.*

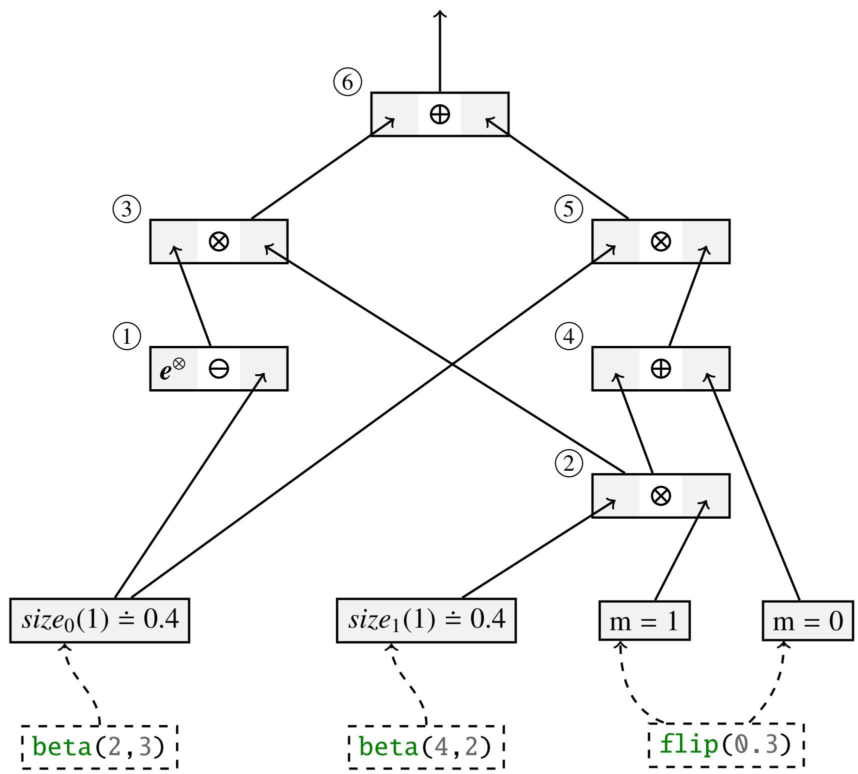

Figure 7.4: At the bottom of the circuit we see the distributions feeding in. The `flip` distribution feeds into its two possible (non-zero probability) outcomes. The two `beta` distributions feed into an observation statement each. We use the '≐' symbol to denote such an observation. Note how we identify each of the two random variables for the size by a unique identifier in their respective subscripts. The circled numbers next to the internal nodes, i.e. the sum and product nodes, will allow us to reference the nodes later on and do not form a part of the algebraic circuit.

The probability of the query (given the evidence) can now be obtained by evaluating recursively the internal nodes in the algebraic circuit using Algorithm 7.21. We perform the evaluation of the circuit in Figure 7.4 for a single iteration of the loop in Algorithm 7.20, and we assume that we have sampled the value $m = 0$ *from the* `flip(0.3)` *distribution.*

$\texttt{Eval}(①)$
$= e^{\otimes} \ominus \alpha_{IALW}(size_0(1) \doteq 0.4)$
$= (1,0) \ominus (1.728, 1)$
$= (1,0)$

$\texttt{Eval}(②)$
$= \alpha_{IALW}(size_1(1) \doteq 0.4) \otimes \alpha_{IALW}(0 = 1)$
$= (0.768, 1) \otimes (0,0)$
$= (0,1)$

$\texttt{Eval}(③)$
$= \texttt{Eval}(①) \otimes \texttt{Eval}(②)$
$= (1,0) \otimes (0,1)$
$= (0,1)$

$\texttt{Eval}(④)$
$= \texttt{Eval}(②) \oplus \alpha_{IALW}(0 = 0)$
$= (0,1) \oplus (1,0)$
$= (1,0)$

$\texttt{Eval}(⑤)$
$= \alpha_{IALW}(1 = 1) \otimes \texttt{Eval}(②)$
$= (1.728, 1) \otimes (1,0)$
$= (1.728, 1)$

$\texttt{Eval}(⑥)$
$= \texttt{Eval}(③) \oplus \texttt{Eval}(⑤)$
$= (0,1) \oplus (1.728, 1)$
$= (1.728, 1)$

If we evalute the circuit for a sample $m = 1$ we obtain in a similar fashion the result $\texttt{Eval}(⑥) = (0.768, 1)$. *Moreover, if we evaluate the circuit multiple times we obtain (in the limit) 70% of the time the outcome* $(1.728, 1)$ *and 30% of the time the value* $(0.768, 1)$. *This yields an average of* $(0.7 \times 1.728, 1) \oplus (0.3 \times 0.768, 1) = (1.440, 1)$ *and represents the unnormalized infinitesimal algebraic likelihood weight of the evidence. The unnormalized infinitesimal algebraic likelihood weight of the query conjoined with the evidence is obtain again in a similar fashion but with the samples for $m = 0$ being discarded. This then yields the result* $(0.3 \times 1.728, 1)$. *Dividing these two (unnormalized) infinitesimal algebraic likelihood weights by each other gives the probability of the query.*

$$
\begin{aligned}
&P((\texttt{m=:=1}) = \top \mid \texttt{size} \doteq 4/10) \\
&= (0.3 \times 1.728, 1) \oslash \big((0.7 \times 0.768, 1) \oplus (0.3 \times 1.728, 1)\big) \\
&= (0.2304/1.440, 1{-}1) \\
&= (0.16, 0)
\end{aligned}
$$

7.5. Partial Symbolic Inference

Evaluating circuits using binary random variables is quite wasteful: on average half of the samples are unused for one of the two possible outcomes (0 or 1). We can remedy this by performing (exact) symbolic inference on binary random variables and replace the comparisons where they appear with their expectation. For instance, we replace `m=:=1` by the infinitesimal number $(0.3, 0)$ instead of sampling a value for `m` and testing whether the sample satisfies the constraint. This technique is also used by other probabilistic programming languages such as DC-ProbLog [Fierens et al., 2015] and Dice [Holtzen et al., 2020]. The main difference to DC-ProbLog is that those languages only support binary random variables (and by extension discrete random variables with finite support), while DC-ProbLog interleaves discrete and continuous random variables.

In a sense, the expectation gets pushed from the root of the algebraic circuit representing a probability to its leaves. This is, however, only possible if the circuit respects specific properties. Namely, the ones respected by d-DNNF formulas (cf. Section 7.4), which we use as our representation language for the probability.

Definition 7.24 (Symbolic IALW Label of a Literal)**.** *Given is an ancestral sample* $\mathbf{s}^{(i)} = (s_1^{(i)}, \dots, s_M^{(i)})$ *for the random variables* $\mathcal{V} = (v_1, \dots, v_M)$*. The Symbolic IALW (SIALW) label of a positive literal* ℓ *is an infinitesimal number given by:*

$$\alpha^{(i)}_{SIALW}(\ell) = \begin{cases} (p_\ell, 0), & \textit{if } \ell \textit{ encodes a probabilistic fact} \\ \alpha^{(i)}_{IALW}(\ell), & \textit{otherwise} \end{cases}$$

For the negated literals we have the following labeling function:

$$\alpha^{(i)}_{SIALW}(\neg\ell) = \begin{cases} (1-p_\ell, 0), & \textit{if } \ell \textit{ encodes a probabilistic fact} \\ \alpha^{(i)}_{IALW}(\neg\ell), & \textit{otherwise} \end{cases}$$

The number p_ℓ *is the label of the probabilistic fact in a DC-ProbLog program.*

In the definition above we replace the label of a comparison that corresponds to a probabilistic fact with the probability of that fact being satisfied. This has already been shown to be beneficial when performing inference, both in terms of inference time and accuracy of Monte Carlo estimates [Zuidberg Dos Martires et al., 2019b]. Following the work of [Kolb et al., 2019] one could also develop more sophisticated methods to detect which comparison in the leaves can be replaced with their expectation. We leave this for future work.

Example 7.25 (Symbolic IALW on Algebraic Circuit)**.** *Symbolic inference for the random variable m from the circuit in Example 7.22 results in annotating the leaf nodes for the different outcomes of the random variable m with the probabilities of the respective outcomes. This can be seen in the red dashed box in the bottom right of Figure 7.5.*

Evaluating the marginalized circuit now returns immediately the unnormalized algebraic model count for the evidence without the need to draw samples and consequently without the need to sum over the samples.

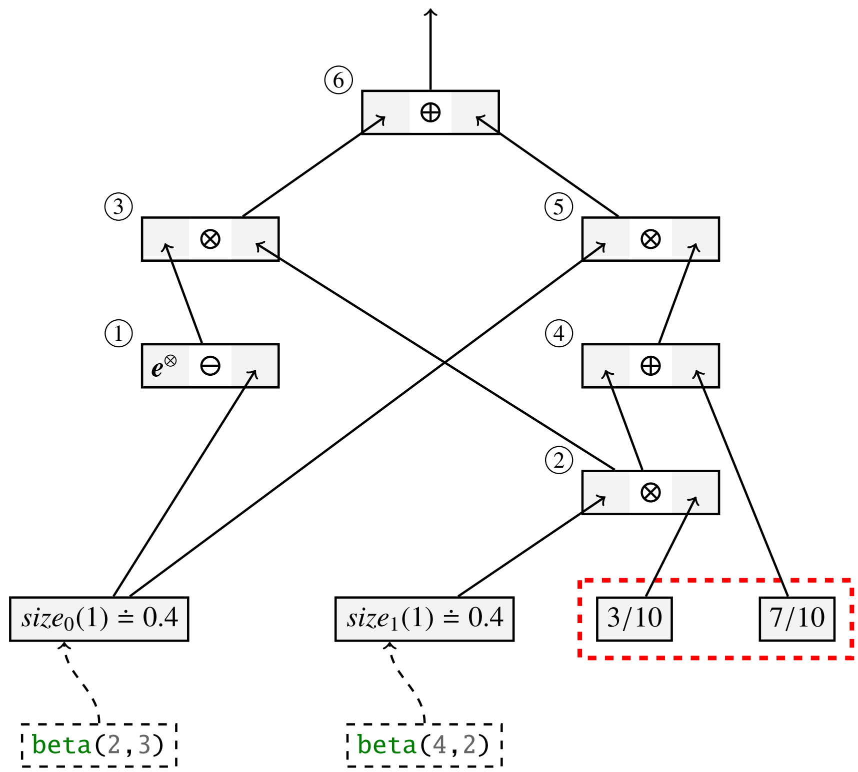

Figure 7.5: Circuit representation of the SIALW algorithm for the probability $P(\texttt{size} \doteq 4/10)$.

$$
\begin{aligned}
&\texttt{Eval}(1)\\
&= e^{\otimes} \ominus \alpha_{SIALW}(size_0(1) \doteq 0.4)\\
&= (1, 0) \ominus (1.728, 1)\\
&= (1, 0)
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{Eval}(2)\\
&= \alpha_{SIALW}(size_1(1) \doteq 0.4) \otimes \alpha_{SIALW}(0.3)\\
&= (0.768, 1) \otimes (0.3, 0)\\
&= (0.2304, 1)
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{Eval}(3)\\
&= \texttt{Eval}(1) \otimes \texttt{Eval}(2)\\
&= (1, 0) \otimes (0.2304, 1)\\
&= (0.2304, 1)
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{Eval}(4)\\
&= \texttt{Eval}(2) \oplus \alpha_{SIALW}(1 = 1)\\
&= (0.2304, 1) \oplus (0.7, 0)\\
&= (0.7, 0)
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{Eval}(5)\\
&= \alpha_{SIALW}(1 = 1) \otimes \texttt{Eval}(2)\\
&= (1.728, 1) \otimes (0.7, 0)\\
&= (1.2096, 1)
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{Eval}(6)\\
&= \texttt{Eval}(3) \oplus \texttt{Eval}(5)\\
&= (0.2304, 1 \oplus (1.2096, 1)\\
&= (1.440, 1)
\end{aligned}
$$

8. DC-ProbLog and the Probabilistic Programming Landscape

In recent years a plethora of different probabilistic programming languages have been developed. We discuss these by pointing key features present in DC-ProbLog (lsited bellow), which are missing in specific related works. We organize these features along the three key contributions stated in Section 1. Our first key contribution is the introduction of the hybrid distribution semantics with the following features:

C1.1 random variables with (possibly) infinite sample spaces

C1.2 functional dependencies between random variables

C1.3 uniform treatment of discrete and continuous random variables

C1.4 negation

Our second contribution is the introduction of the DC-ProbLog language, which

C2.1 has purely discrete PLPs and their semantics as a special case,

C2.2 supports a rich set of comparison predicates, and

C2.3 is a Turing complete language (DC-PLP)

Our last contributions concern inference, which includes

C3.1 a formal definition of the hybrid probabilistic inference task inference task,

C3.2 an inference algorithm called IALW,

C3.3 that uses standard knowledge compilation in the hybrid domain.

8.1. ProbLog and Distributional Clauses

The DC-ProbLog language is a generalization of ProbLog, both in terms of syntax and semantics. A DC-ProbLog program that does not use distributional clauses (or distributional facts) is also a ProbLog program, and both views define the same distribution over the logical vocabulary of the program. DC-ProbLog properly generalizes ProbLog to include random variables with infinite sample spaces (C1.1).

On a syntactical level, DC-ProbLog is closely related to the Distributional Clauses (DC) language, with which it shares the ~/2 predicate used in infix notation. In Appendix E we discuss in more detail the relationship between DC-ProbLog and the Distributional Clauses language. Concretely, we point out that DC-ProbLog generalizes the original and negation-free version of DC [Gutmann et al., 2011] (C1.4). However, DC-ProbLog differs in its declarative interpretation of negation from the procedural interpretation as introduced to DC by Nitti et al. [2016]. As a consequence, the semantics of DC and ProbLog differ in the absence of continuous random variables, while DC-ProbLog is a strict generalization of ProbLog (C2.1).

8.2. *Extended PRISM*

An early attempt of equipping a probabilistic logic programming language with continuous random variables can be found in [Islam et al., 2012], which was dubbed *Extended PRISM*. Similar to DC-ProbLog, Extended PRISM's semantics are based again on Sato's distribution semantics. However, Extended PRISM assumes, just like Distributional Clauses, pairwise mutually exclusive proofs (we refer again to Appendix E for details on this). On the expressivity side, Extended PRISM only supports linear equalities – in contrast to DC-ProbLog, where also inequalities are included in the semantics of the language (C2.2). An advantage of restricting possible constraints to equalities is the possibility of performing exact symbolic inference. In this regard, Extended PRISM, together with its symbolic inference algorithm, can be viewed as a logic programming language that has access to a computer algebra system. Swapping out the approximate Sampo-inspired inference algorithm in DC-ProbLog by an exact inference algorithm using symbolic expression manipulations would result in an inference approach closely related to that of Extended PRISM. One possibility would be to use the Symbo algorithm presented in [Zuidberg Dos Martires et al., 2019b], which uses the PSI-language [Gehr et al., 2016] as its (probabilistic) computer algebra system.

8.3. *Probabilistic Constraint Logic Programming*

Impressive work on extending probabilistic logic programs with continuous random variables was presented by Michels et al. [2015] with the introduction of Probabilistic Constraint Logic Programming (PCLP). The semantics of PCLP are again based on Sato's distribution semantics and the authors also presented an approximate inference algorithm for hybrid probabilistic logic programs. Interestingly, the algorithm presented in [Michels et al., 2015] to perform (conditional) probabilistic inference extends weighted model counting to continuous random variables using imprecise probabilities, and more specifically credal sets.

A shortcoming of PCLP's semantics is the lack of direct support for generative definitions of random variables, i.e., random variables can only be interpreted within constraints, but not within distributions of other random variables as is possible in DC-ProbLog (C1.2). Azzolini et al. [2021] define a non-credal version of this semantics using a product measure over a space that explicitly separates discrete and continuous random variables, assuming that a measure over the latter is given as part of the input without further discussion of how this part of the measure is specified in a program. Furthermore, they do not define any inference tasks (C3.1), e.g. computing conditional probabilities (cf. Section 5), nor do they provide an inference algorithm (C3.2).

A later proposal for the syntax of such programs [Azzolini and Riguzzi, 2021] combines two classes of terms (logical and continuous ones) with typed predicates and functors, and defines mixture variables as well as arithmetic expressions over random variables through logical clauses. In other words, user-defined predicates define families of random variables through the use of typed arguments of the predicate identifying a specific random variable, arguments providing parameters for the distribution, and one argument representing the random variable itself. In contrast, the syntax of DC-ProbLog clearly identifies all random variables through explicit terms introduced through distributional facts or distributional clauses, explicitly exposes the probabilistic dependency structure by using random variable terms inside distribution terms, and avoids typing through argument positions. Moreover, DC-ProbLog takes a uniform view on all random variables in terms of semantics, thereby avoiding treating discrete and continuous random variables separately (C1.3).

8.4. BLOG

Notable in the domain of probabilistic logic programming is also the BLOG language [Milch et al., 2005a, Wu et al., 2018]. Contrary to the aforementioned probabilistic logic programming languages, BLOG's semantics are not specified using Sato's distribution semantics but via so-called *measure-theoretic Bayesian networks* (MTBN), which were introduced in [Wu et al., 2018]. MTBNs can be regarded as the assembly language for BLOG: every BLOG program is translated or compiled to an MTBN. With DC-ProbLog we follow a similar pattern: every DC-ProbLog program with syntactic sugar (e.g. annotated disjunctions) is transformed into DF-PLP program. The semantics are defined on the bare-bones program. Note that the assembly language for DC-ProbLog (DF-PLP) is Turing complete. This is not the case for MTBNs (C2.3).

8.5. Non-logical Probabilistic Programming

As first pointed out by Russell [2015] and later on elaborated upon by Kimmig and De Raedt [2017], probabilistic programs fall either into the *possible worlds semantics* category or the *probabilistic execution traces semantics* category. The former is usually found in logic based languages, while the latter is the prevailing view in imperative and functional probabilistic languages.

While, the probabilistic programming languages discussed so far follow the possible worlds paradigm, many languages follow the execution traces paradigm, either as a probabilistic functional language [Goodman et al., 2008, Wood et al., 2014] or as a imperative probabilistic language [Gehr et al., 2016, Salvatier et al., 2016, Carpenter et al., 2017, Bingham et al., 2019, Ge et al., 2018]. Generally speaking, functional and imperative probabilistic programming languages target first and foremost continuous random variables, and discrete random variables are only added as an afterthought. A notable exception is the functional probabilistic programming language Dice [Holtzen et al., 2020], which targets discrete random variables exclusively.

Concerning inference in probabilistic programming, we can observe general trends in logical and non-logical probabilistic languages. While the latter are interested in adapting and speeding up approximate inference algorithms, such as Markov chain Monte Carlo sampling schemes or variational inference, the former type of languages are more invested in exploiting independences in the probabilistic programs, mainly by means of knowledge compilation. Clearly, these trends are not strict. For instance, Obermeyer et al. [2019] proposed so-called *funsors* to express and exploit independences in Pyro [Bingham et al., 2019], an imperative probabilistic programming language, and Gehr et al. [2016] developed a computer algebra system to perform exact symbolic probabilistic inference.

8.6. Representation of Probabilistic Programs at Inference Time

Lastly, we would like to point out a key feature of the IALW inference algorithm that sets it apart from any other inference scheme for probabilistic programming in the hybrid domain. But first, let us briefly talk about computing probabilities in probabilistic programming. Roughly speaking, probabilities are computed summing and multiplying weights. These can for example be represented as floating point numbers or symbolic expressions. The collection of all operations that were performed to obtain the probability of a query to a program is called the computation graph. Now, the big difference between IALW and other inference algorithms lies in the structure of the computation graph. IALW represents the computation graph as a directed acyclic graph (DAG), while all other languages, except some purely discrete languages [Fierens et al., 2015, Holtzen et al., 2020], use a tree representation. IALW is the first inference algorithm in the discrete-continuous domain that uses DAGs (C3.3)! In cases where the computation graph can be represented as a DAG the size of the representation might be exponentially smaller compared to tree representations, which leads to faster inference times.

Note that Gutmann et al. [2010] and more recently Saad et al. [2021] presented implementations of hybrid languages where the inference algorithm leverages directed acyclic graphs, as well. However, the constraints that may be imposed on random variables are limited to univariate equalities and inequalities. In the weighted model integration literature it was shown that such probability computations can be mapped to probability computations of discrete random variables only [Zeng and Van den Broeck, 2020].

9. Conclusions

We introduced DC-ProbLog, a hybrid PLP language for the discrete-continuous domain and its accompanying hybrid distribution semantics. DC-ProbLog strictly extends the discrete ProbLog language [De Raedt et al., 2007, Fierens et al., 2015] and the negation-free Distributional Clauses [Gutmann et al., 2011] language. In designing the language and its semantics we adapted Poole [2010]'s design principle of percolating probabilistic logic programs into two separate layers: the random variables and the logic program. Boolean comparison atoms then form the link between the two layers. It is this clear separation between the random variables and the logic program that has allowed us to use simpler language constructs and to write programs using a more concise and intuitive syntax than alternative hybrid PLP approaches [Gutmann et al., 2010, Nitti et al., 2016, Speichert and Belle, 2019, Azzolini et al., 2021].

Separating random variables from the logic program also allowed us to develop the IALW algorithm to perform inference in the hybrid domain. IALW is the first algorithm based on knowledge compilation and algebraic model counting for hybrid probabilistic programming languages and as such it generalizes the standard knowledge compilation based approach for PLP. It is noteworthy that IALW correctly computes conditional probabilities in the discrete-continuous domain using the newly introduced infinitesimal numbers semiring.

Interesting future research directions include adapting ideas from functional probabilistic programming (the other declarative programming style besides logic programming) in the context of probabilistic logic programming. For instance, extending DC-ProbLog with a type system [Schrijvers et al., 2008] or investigating more recent advances, such as *quasi-Borel spaces* [Heunen et al., 2017] in the context of the distribution semantics.

Acknowledgement

This research received funding from the Wallenberg AI, Autonomous Systems and Software Program (WASP) of the Knut and Alice Wallenberg Foundation, the Flemish Government (AI Research Program), the KU Leuven Research Fund, the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No [694980] SYNTH: Synthesising Inductive Data Models), and the Research Foundation - Flanders.

References

Damiano Azzolini and Fabrizio Riguzzi. Syntactic requirements for well-defined hybrid probabilistic logic programs. In *Technical communications of ICLP 2021*, 2021.

Damiano Azzolini, Fabrizio Riguzzi, and Evelina Lamma. A semantics for hybrid probabilistic logic programs with function symbols. *Artificial Intelligence*, 294, 2021.

Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.

Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul A. Szerlip, Paul Horsfall, and Noah D. Goodman. Pyro: Deep universal probabilistic programming. *J. Mach. Learn. Res.*, 20:28:1–28:6, 2019. URL `http://jmlr.org/papers/v20/18-403.html`.

Bob Carpenter, Andrew Gelman, Matthew D Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. Stan: A probabilistic programming language. *Journal of statistical software*, 76(1), 2017.

George Casella and Christian P Robert. Post-processing accept-reject samples: recycling and rescaling. *Journal of Computational and Graphical Statistics*, 7(2): 139–157, 1998.

Natalia Cherchago, Pascal Hitzler, and Steffen Hölldobler. Decidability under the well-founded semantics. In *International Conference on Web Reasoning and Rule Systems*, pages 269–278. Springer, 2007.

Adnan Darwiche. On the tractable counting of theory models application to truth maintenance and belief revision. *Journal of Applied Non-Classical Logics*, 2001.

Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002.

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

Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. Problog: A probabilistic prolog and its application in link discovery. In *IJCAI*, volume 7, pages 2462–2467. Hyderabad, 2007.

Luc De Raedt, Kristian Kersting, Sriraam Natarajan, and David Poole. *Statistical Relational Artificial Intelligence: Logic, Probability, and Computation*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2016.

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

Robert Fung and Kuo-Chu Chang. Weighing and integrating evidence for stochastic simulation in bayesian networks. *Machine Intelligence and Pattern Recognition*, 10: 209–219, 1990.

Hong Ge, Kai Xu, and Zoubin Ghahramani. Turing: A language for flexible probabilistic inference. In *International Conference on Artificial Intelligence and Statistics*, pages 1682–1690, 2018.

Timon Gehr, Sasa Misailovic, and Martin Vechev. PSI: Exact symbolic inference for probabilistic programs. In *International Conference on Computer Aided Verification*. Springer, 2016.

Noah D Goodman, Vikash K Mansinghka, Daniel Roy, Keith Bonawitz, and Joshua B Tenenbaum. Church: a language for generative models. In *Proceedings of the Twenty-Fourth Conference on Uncertainty in Artificial Intelligence*, pages 220–229, 2008.

Bernd Gutmann, Manfred Jaeger, and Luc De Raedt. Extending problog with continuous distributions. In *International Conference on Inductive Logic Programming*. Springer, 2010.

Bernd Gutmann, Ingo Thon, Angelika Kimmig, Maurice Bruynooghe, and Luc De Raedt. The magic of logical inference in probabilistic programming. *Theory and Practice of Logic Programming*, 11(4-5):663–680, 2011.

Chris Heunen, Ohad Kammar, Sam Staton, and Hongseok Yang. A convenient category for higher-order probability theory. In *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–12. IEEE, 2017.

Steven Holtzen, Guy Van den Broeck, and Todd Millstein. Dice: Compiling discrete probabilistic programs for scalable inference. *arXiv preprint arXiv:2005.09089*, 2020.

Muhammad Asiful Islam, CR Ramakrishnan, and IV Ramakrishnan. Inference in probabilistic logic programs with continuous random variables. *Theory and Practice of Logic Programming*, 12(4-5):505–523, 2012.

Kenneth E Iverson. A programming language. In *Proceedings of the May 1-3, 1962, spring joint computer conference*, 1962.

Jules Jacobs. Paradoxes of probabilistic programming: And how to condition on events of measure zero with infinitesimal probabilities. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–26, 2021.

Tomi Janhunen. Representing normal programs with clauses. In *Proceedings of the 16th European Conference on Artificial Intelligence*, pages 358–362, 2004.

Edwin T Jaynes. *Probability theory: The logic of science*. Cambridge university press, 2003.

Joseph B Kadane. *Principles of uncertainty*. CRC Press, 2011.

Herman Kahn. Random sampling (monte carlo) techniques in neutron attenuation problems. i. *Nucleonics (US) Ceased publication*, 6(See also NSA 3-990), 1950.

Kristian Kersting and Luc De Raedt. Bayesian logic programs. In *Proceedings of the 10th International Conference on Inductive Logic Programming, Work-in-progress reports*, pages 1–18, 2000.

Angelika Kimmig and Luc De Raedt. Probabilistic logic programs: Unifying program trace and possible world semantics. In *Workshop on probabilistic programming semantics*, 2017.

Angelika Kimmig, Guy Van den Broeck, and Luc De Raedt. Algebraic model counting. *Journal of Applied Logic*, 2017.

Tuen Kloek and Herman K Van Dijk. Bayesian estimates of equation system parameters: an application of integration by monte carlo. *Econometrica: Journal of the Econometric Society*, pages 1–19, 1978.

Samuel Kolb, Pedro Zuidberg Dos Martires, and Luc De Raedt. How to exploit structure while solving weighted model integration problems. In *Uncertainty in Artificial Intelligence*, pages 744–754. PMLR, 2019.

John W Lloyd. *Foundations of logic programming*. Springer Science & Business Media, 2012.

Vikash Mansinghka, Daniel Selsam, and Yura Perov. Venture: a higher-order probabilistic programming platform with programmable inference. *arXiv preprint arXiv:1404.0099*, 2014.

Theofrastos Mantadelis and Gerda Janssens. Dedicated tabling for a probabilistic setting. In *Technical Communications of the 26th International Conference on Logic Programming*, 2010.

Steffen Michels, Arjen Hommersom, Peter JF Lucas, and Marina Velikova. A new probabilistic constraint logic programming language based on a generalised distribution semantics. *Artificial Intelligence*, 228:1–44, 2015.

Brian Milch, Bhaskara Marthi, Stuart Russell, David Sontag, Daniel L Ong, and Andrey Kolobov. Blog: Probabilistic models with unknown objects. In *19th International Joint Conference on Artificial Intelligence, IJCAI 2005*, pages 1352–1359, 2005a.

Brian Milch, Bhaskara Marthi, David Sontag, Stuart Russell, Daniel L Ong, and Andrey Kolobov. Approximate inference for infinite contingent bayesian networks. In *International Workshop on Artificial Intelligence and Statistics*, pages 238–245. PMLR, 2005b.

Brian Christopher Milch. *Probabilistic Models with Unknown Objects*. PhD thesis, University of California, Berkeley, 2006.

Davide Nitti, Tinne De Laet, and Luc De Raedt. Probabilistic logic programming for hybrid relational domains. *Machine Learning*, 103(3):407–449, 2016.

Fritz Obermeyer, Eli Bingham, Martin Jankowiak, Du Phan, and Jonathan Chen. Functional tensors for probabilistic programming. *arXiv:1910.10775*, 2019.

David Poole. Probabilistic horn abduction and bayesian networks. *Artificial intelligence*, 64(1):81–129, 1993.

David Poole. Probabilistic programming languages: Independent choices and deterministic systems. *Heuristics, probability and causality: A tribute to Judea Pearl*, pages 253–269, 2010.

Fabrizio Riguzzi. *Foundations of Probabilistic Logic Programming*. River Publishers, 2018.

Fabrizio Riguzzi and Terrance Swift. The pita system: Tabling and answer subsumption for reasoning under uncertainty. *Theory and Practice of Logic Programming*, 11(4-5):433–449, 2011.

Stuart Russell. Unifying logic and probability. *Communications of the ACM*, 58(7): 88–97, 2015.

Feras A Saad, Martin C Rinard, and Vikash K Mansinghka. Sppl: Probabilistic programming with fast exact symbolic inference. In *Conference on Programming Language Design and Implementation*, 2021.

John Salvatier, Thomas V Wiecki, and Christopher Fonnesbeck. Probabilistic programming in python using pymc3. *PeerJ Computer Science*, 2:e55, 2016.

Taisuke Sato. A statistical learning method for logic programs with distribution semantics. In *In Proceedings of the 12th International Conference On Logic Programming (ICLP'95*. Citeseer, 1995.

Taisuke Sato and Yoshitaka Kameya. Prism: a language for symbolic-statistical modeling. In *IJCAI*, volume 97, pages 1330–1339, 1997.

Tom Schrijvers, Vítor Santos Costa, Jan Wielemaker, and Bart Demoen. Towards typed prolog. In *ICLP*, 2008.

Chung-chieh Shan and Norman Ramsey. Exact Bayesian inference by symbolic disintegration. In *ACM SIGPLAN Notices*, volume 52, pages 130–144. ACM, 2017.

Stefanie Speichert and Vaishak Belle. Learning probabilistic logic programs over continuous data. In *International Conference on Inductive Logic Programming*, 2019.

Sam Staton, Frank Wood, Hongseok Yang, Chris Heunen, and Ohad Kammar. Semantics for probabilistic programming: higher-order functions, continuous distributions, and soft constraints. In *2016 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–10. IEEE, 2016.

Leslie G Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 1979.

Allen Van Gelder, Kenneth A Ross, and John S Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM (JACM)*, 38(3):619–649, 1991.

Joost Vennekens, Sofie Verbaeten, and Maurice Bruynooghe. Logic programs with annotated disjunctions. In *International Conference on Logic Programming*, pages 431–445. Springer, 2004.

Jonas Vlasselaer, Joris Renkens, Guy Van den Broeck, and Luc De Raedt. Compiling probabilistic logic programs into sentential decision diagrams. In *Proceedings Workshop on Probabilistic Logic Programming (PLP)*, pages 1–10, 2014.

Frank Wood, Jan Willem van de Meent, and Vikash Mansinghka. A new approach to probabilistic programming inference. In *Proceedings of the 17th International conference on Artificial Intelligence and Statistics*, pages 1024–1032, 2014.

Yi Wu, Siddharth Srivastava, Nicholas Hay, Simon Du, and Stuart Russell. Discrete-continuous mixtures in probabilistic programming: Generalized semantics and inference algorithms. In *International Conference on Machine Learning*, pages 5339–5348, 2018.

Zhe Zeng and Guy Van den Broeck. Efficient search-based weighted model integration. In *Uncertainty in Artificial Intelligence*, pages 175–185. PMLR, 2020.

Pedro Zuidberg Dos Martires. *From Atoms to Possible Worlds: Probabilistic Inference in the Discrete-Continuous Domain*. KU Leuven, 2020.

Pedro Zuidberg Dos Martires, Vincent Derkinderen, Robin Manhaeve, Wannes Meert, Angelika Kimmig, and Luc De Raedt. Transforming probabilistic programs into algebraic circuits for inference and learning. In *Transformations for Machine Learning Workshop*, 2019a.

Pedro Zuidberg Dos Martires, Anton Dries, and Luc De Raedt. Exact and approximate weighted model integration with probability density functions using knowledge compilation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 7825–7833, 2019b.

A. Logic Programming

We briefly summarize key concepts of the syntax and semantics of logic programming; for a full introduction, we refer to [Lloyd, 2012].

A.1. Building Blocks

The basic building blocks of logic programs are *variables* (denoted by strings starting with upper case letters), *constants*, *functors* and *predicates* (all denoted by strings starting with lower case letters). A *term* is a variable, a constant, or a functor f of *arity* n followed by n terms t_i, i.e., $f(t_1, \ldots, t_n)$. An *atom* is a predicate p of arity n followed by n terms t_i, i.e., $p(t_1, \ldots, t_n)$. A predicate p of arity n is also written as p/n. A *literal* is an atom or a negated atom $not(p(t_1, \ldots, t_n))$.

A.2. Logic Programs

A *definite clause* is a universally quantified expression of the form $h :- b_1, \ldots, b_n$ where h and the b_i are atoms. h is called the *head* of the clause, and $b_1, \ldots, b_n$ its *body*. Informally, the meaning of such a clause is that if all the b_i are true, h has to be true as well. A *normal clause* is a universally quantified expression of the form $h :- l_1, \ldots, l_n$ where h is an atom and the l_i are literals. If $n = 0$, a clause is called *fact* and simply written as h. A *definite clause program* or *logic program* for short is a finite set of definite clauses. A *normal logic program* is a finite set of normal clauses.

A.3. Substitutions

A *substitution* θ is an expression of the form $\{V_1/t_1, \ldots, V_m/t_m\}$ where the V_i are different variables and the t_i are terms. Applying a substitution θ to an expression e (term or clause) yields the *instantiated* expression $e\theta$ where all variables V_i in e have been simultaneously replaced by their corresponding terms t_i in θ. If an expression does not contain variables it is *ground*. Two expressions e_1 and e_2 can be *unified* if and only if there are substitutions θ_1 and θ_2 such that $e_1\theta_1 = e_2\theta_2$.

A.4. Herbrand Universe

The *Herbrand universe* of a logic program is the set of ground terms that can be constructed using the functors and constants occurring in the program. The *Herbrand base* of a logic program is the set of ground atoms that can be constructed from the predicates in the program and the terms in its Herbrand universe. A truth value assignment to all atoms in the Herbrand base is called *Herbrand interpretation*, and is also represented as the set of atoms that are true according to the assignment. A Herbrand interpretation is a *model* of a clause $h :- b_1, \ldots, b_n$. if for every substitution θ such that all $b_i\theta$ are in the interpretation, $h\theta$ is in the interpretation as well. It is a model of a logic program if it is a model of all clauses in the program. The model-theoretic semantics of a definite clause program is given by its smallest Herbrand model with respect to set inclusion, the so-called *least Herbrand model* (which is unique). We say that a logic program $\mathcal{P}$ *entails* an atom a, denoted $\mathcal{P} \models a$, if and only if a is true in the least Herbrand model of $\mathcal{P}$.

B. Table of Notations

symbol	meaning	for details, see
Δ	set of distribution functors	Definition 3.1
Φ	set of arithmetic functors	Definition 3.1
Π	set of comparison predicates	Definition 3.1
Ω_ν	sample space of random variable ν	Definition 3.4
$\omega(\cdot)$	value assignment function	Definition 3.4
$\mathcal{D}$	distributional database	Definition 3.5
$\mathcal{V}$	set of random variables	Definition 3.5
$P_{\mathcal{V}}$	probability measure over $\mathcal{V}$ defined by $\mathcal{D}$	Proposition 3.12
$\mathcal{F}$	set of Boolean comparison atoms	Definition 3.13
$P_{\mathcal{F}}$	probability measure over interpretations of $\mathcal{F}$ induced by $P_{\mathcal{V}}$	Proposition 3.15
$\mathcal{P}^{DF} = \mathcal{D} \cup \mathcal{R}$	DF-PLP program	Definition 3.16
$\mathcal{F}_{\omega(\mathcal{V})}$	consistent comparison database induced by ω on the random variables in $\mathcal{V}$	Definition 3.18
$P_{\mathcal{P}^{DF}}$	probability measure over Herbrand interpretations defined by $\mathcal{P}^{DF}$	Proposition 3.20
$\mathcal{P}$	DC-ProbLog program	Definition 4.12
$\mathcal{P}^*$	AD-free DC-ProbLog program	Definition 4.15
$\mathcal{H}_{\mathcal{P}^*}$	set of heads of distributional clauses in $\mathcal{P}^*$	Definition 4.15
$\mathcal{T}_{\mathcal{P}^*}$	random terms in $\mathcal{H}_{\mathcal{P}^*}$	Definition 4.15
$C_{\mathcal{P}^*}$	set of distributional clauses in $\mathcal{P}^*$	Definition 4.20
$K(\cdot)$	contextualization function	Definition 4.24
$\mathcal{P}^{DF,*}$	DF-PLP program providing the semantics of $\mathcal{P}^*$	Definition 4.25
$MOD(\mathcal{P})$	models of a program $\mathcal{P}$	Theorem 6.4
$ENUM(\phi)$	models of a propositional formula ϕ	Theorem 6.4
$\alpha(\cdot)$	labeling function of a propositional literal	Definition 6.5
$[\![\cdot]\!]$	Iverson bracket denoting an indicator function	Definition 6.5
$\mathbb{E}[\cdot]$	expected value	Theorem 6.9
$\mathbb{I}$	set of infinitesimal numbers	Equation 7.2
$\mathcal{S}$	set of ancestral samples	Equation 7.11

C. Proofs of Propositions in Section 3

C.1. *Proof of Proposition 3.12*

Proposition 3.12. *A well-defined distributional database $\mathcal{D}$ defines a unique probability measure $P_{\mathcal{V}}$ on value assignments $\omega(\mathcal{V})$.*

Proof. The proof is analogous to that for the semantics of well-defined Bayesian Logic Programs (BLPs) [Kersting and De Raedt, 2000, Theorem 4.9]. They show that such a probability measure exists over a non-empty set of random variables if the ancestor structure of the random variables is acyclic and every random variable has a finite set of ancestors, which are exactly conditions W2 and W1 in Definition 3.10. The key idea is that under these conditions, for each finite subset of random variables closed under the ancestor relation, the joint distribution on that set has the form of a Bayesian network, and factorizes into the product of the individual variables' distributions. This family of distributions forms the basis of the unique measure over the potentially infinite set $\mathcal{V}$. We refer to [Kersting and De Raedt, 2000, Theorem 4.9] for technical details. □

Note that while BLPs also use LP syntax to define the random variables and structure of a Bayesian network, the way they use that syntax is fundamentally different from ours.

C.2. Proof of Proposition 3.15

Proposition 3.15. *The probability measure $P_{\mathcal{V}}$, defined by a well-defined distributional database $\mathcal{D}$, induces a unique probability measure $P_{\mathcal{F}}$ over value assignments to the comparison atoms $\mathcal{F}$.*

Proof. To show existence of the measure $P_{\mathcal{F}}$ (i.e. the basic distribution), we fix an arbitrary enumeration $\mu_1, \mu_2, \ldots,$ of the atoms in $\mathcal{F}$. Each μ_i *depends* on a finite set $\mathcal{V}_i \subseteq \mathcal{V}$ of random variables, namely those mentioned in μ_i, as well as their ancestor sets. We write $\mathcal{V}_{\leq n} = \bigcup_{1\leq j\leq n} \mathcal{V}_j$ for the union of random variables that the first n atoms in the enumeration depend on. By $P_{\mathcal{V}_{\leq n}}$ we denote the measure restricted to this set.

By definition, all queries $\mu_i \in \mathcal{F}$ are Lebesgue-measurable, and we thus get a family of distributions

$$P_{\mathcal{F}}^{(n)}(\mu_1 = b_1, \ldots, \mu_n = b_n) = \int_{\Omega(\mathcal{V}_{\leq n})} \mathbf{1}_{[\mu_1=b_1\wedge\ldots\wedge\mu_n=b_n]}(\omega(\mathcal{V}_{\leq n}))\, dP_{\mathcal{V}_{\leq n}} \qquad \text{(C.1)}$$

where the b_i belong to the set $\{\bot, \top\}$, $P_{\mathcal{V}_{\leq n}}$ factorizes over the random variables in $\mathcal{V}_{\leq n}$, $\Omega(\mathcal{V}_{\leq n})$ denotes the space of possible assignments for variables in $\mathcal{V}_{\leq n}$, and $\mathbf{1}_{[\varphi]}$ is the indicator function, i.e., equals 1 if φ is true and 0 otherwise. The definition in terms of an indicator function and the measurability of the underlying Boolean queries ensures that this family of distributions is of the form required for the distribution semantics, i.e. they are well-defined probability distributions satisfying the compatibility condition: $P_{\mathcal{F}}^{(n)}$ can be obtained from $P_{\mathcal{F}}^{(n+1)}$ by summing out μ_{n+1}. There thus exists a completely additive probability measure $P_{\mathcal{F}}$ over the space of truth value assignments to $\mathcal{F}$ such that for any n, we have

$$P_{\mathcal{F}}(\mu_1 = b_1, \ldots, \mu_n = b_n) = P_{\mathcal{F}}^{(n)}(\mu_1 = b_1, \ldots, \mu_n = b_n) \qquad \text{(C.2)}$$

□

C.3. Proof of Proposition 3.20

Proposition 3.20. *A valid DF-PLP program $\mathcal{P}^{DF}$ induces a unique probability measure $P_{\mathcal{P}^{DF}}$ over Herbrand interpretations.*

Proof. To show this, we consider two cases. If $\mathcal{D}$ is empty, i.e. $\mathcal{P}^{DF}$ does not define any random variables, the semantics of $\mathcal{P}^{DF}$ is the well-founded model of $\mathcal{R}$. Thus, normal logic programs (with total well-founded models) are a special case of DF-PLP.

If $\mathcal{D}$ is not empty, we follow Sato's construction to obtain the probability measure $P_{\mathcal{P}^{DF}}$ over Herbrand interpretations from $P_{\mathcal{F}}$. To do so, we fix an enumeration $\mu_1, \mu_2, \dots$ of all atoms in the Herbrand base, which includes those in $\mathcal{F}$. As $\mathcal{P}^{DF}$ is valid, for every consistent comparison database $\mathcal{F}_{\omega(\mathcal{V})}$ (cf. Definition 3.18), the logic program $\mathcal{F}_{\omega(\mathcal{V})} \cup \mathcal{R}$ has a total well-founded model $M_{\omega(\mathcal{V})}$, and we can thus define

$$P^{(n)}_{\mathcal{P}^{DF}}(\mu_1 = b_1, \dots, \mu_n = b_n) := P_{\mathcal{F}}(\{\mathcal{F}_\omega \mid M_{\omega(\mathcal{V})} \models \mu_1^{b_1} \wedge \dots \wedge \mu_n^{b_n}\}) \tag{C.3}$$

where, $b_i \in \{\bot, \top\}$, $\mu_i^1 = \mu_i$, and $\mu_i^0 = \neg\mu_i$. It follows again that there is a completely additive probability measure $P_{\mathcal{P}^{DF}}$ over Herbrand interpretations. □

D. Beyond Mixtures

By definition, we impose on distributional clauses mutual exclusivity of their bodies when they share a random term (cf. Definition 4.20). That is, if we have a set of distributional clauses: $\{\tau \sim \delta_i :- \beta_1 \dots, \tau \sim \delta_n :- \beta_n\}$ we impose that the conjunction of two distinct bodies β_i and β_j ($i \neq j$) is false.

A further condition that we might impose, which is, however, not necessary to define a valid distributional clause, is exhaustiveness. Let us consider again the set of distributional clauses $\{\tau \sim \delta_i :- \beta_1 \dots, \tau \sim \delta_n :- \beta_n\}$. We call this set exhaustive if the disjunction of all the β_i's is equivalent to true.

A set of exhaustive distributional clauses can be interpreted as a mixture models as they assign a unique distribution to the random term in any possible context. When, the bodies of such distributional clauses are not exhaustive, however, they may interact with the logic program in rather subtle ways, especially if negation is involved. We demonstrate this in the examples below.

Example D.1. *Consider the following program fragments*

```
q :- not (x=:=1).
```

and

```
aux :- x=:=1.
q :- not aux.
```

and now assume `x` *follows a mixture distribution, e.g.,*

```
0.2::b.
x~flip(0.5) :- b.
x~flip(0.9) :- not b.
```

With such a mixture model, as in the case of a distributional fact, "`x` *has an associated distribution" is always true, and both fragments agree on the truth value of* `q`.

In general, however, only the first of these two conditions is necessary, and it is thus possible to associate a distribution with a random term in *some* contexts only.

Example D.2. *Consider again the two program fragments above with the following non-exhaustive definition of* `x`*:*

```
0.2::b.
x~flip(0.5) :- b.
```

With this definition, "`x` *has an associated distribution" is true if and only if* `b` *is true, and the two fragments therefore no longer agree on the truth values of* `q`*, as we more easily see after eliminating the distributional clause. We omit the transformation of the probabilistic fact for brevity. The fragment defining* `x` *transforms to*

```
v1~flip(0.5).
rv(x,v1) :- b.
```

The first program fragment maps to

```
q :- rv(x,v1), not (v1=:=1).
```

and the second one to

```
aux :- rv(x,v1), v1=:=1.
q :- not aux.
```

which clearly exposes the difference in how the negation is interpreted.

As this example illustrates, if random variables are defined through non-exhaustive sets of DCs, we can no longer refactor the logic program independently of the definition of the random variables in general, as it interacts with the context structure. The reason is that DC-ProbLog's declarative semantics builds upon the principle that the distributional database is declared *independently* of the logic program, and can thus be combined modularly and declaratively with *any* logic program over its comparison atoms. This is no longer the case with such arbitrary sets of DCs, which intertwine the definition of the two parts of a DF-PLP program. We note that this differs from the procedural view on the existence of random variables taken in the Distributional Clauses language [Nitti et al., 2016], as we discuss in more detail in Appendix E.

E. Relation to the DC language

Distributional clauses were first introduced in the language of the same name by Gutmann et al. [2011], which at that point did not support negation. For negation-free programs, our interpretation of distributional clauses exactly corresponds to theirs, and DC-ProbLog thus generalizes both ProbLog (with negation) and the original (definite) distributional clause language.

In the following, we first discuss how the semantics of DC-ProbLog differs from Nitti et al. [2016]'s procedural view on negated comparison atoms, and then how DC-ProbLog's acyclicity conditions imposed on valid programs differ from those of Gutmann et al. [2011].

E.1. Non-exhaustive sets of DCs

Nitti et al. [2016] have extended the procedural view of the stochastic T_P operator to locally stratified programs with negation under the perfect models semantics.[6] In their view, a distributional clause `x~d :- body` is informally interpreted as "if `body` is true, define a random variable `x` with distribution `d`". They then use the principle that "any comparison involving a non-defined variable will fail; therefore, its negation will succeed", i.e., they apply negation as failure to comparison atoms. In contrast, as already illustrated in Section D, we take a purely declarative view here, where all random variables are defined up front, independently of logical reasoning, and distributional clauses serve as syntactic sugar to compactly talk about a group of random variables. Then, truth values of comparison atoms are fully determined by their external interpretation, and do not involve reasoning about whether a random variable is defined or not. That is, we apply classical negation to comparison atoms, and restrict negation as failure to atoms defined by the logic program itself.

The following example adapted from Nitti et al. [2016] illustrates the difference.

Example E.1. *Consider the following program about the color of certain objects, where the number of objects is given by the random variable* `n`*:*

```
n ~ uniform([1,2,3]).
color(1) ~ uniform([red,green,blue]) :- 1=<n .
color(2) ~ uniform([red,green,blue]) :- 2=<n .
color(3) ~ uniform([red,green,blue]) :- 3=<n .
not_red :- not color(2)=:=red .
not_red_either :- color(2)=\=red.
```

The DC-ProbLog semantics is given by the transformed program:

```
v0 ~ uniform([1,2,3]).
v1 ~ uniform([red,green,blue]).
v2 ~ uniform([red,green,blue]).
v3 ~ uniform([red,green,blue]).

rv(n,v0).
rv(color(1),v1) :- rv(n,v0), 1=<v0.
rv(color(2),v2) :- rv(n,v0), 2=<v0.
rv(color(3),v3) :- rv(n,v0), 3=<v0.

not_red :- rv(color(2),v2), not v2=:=red.
not_red_either :- rv(color(2),v2), v2=\=red.
```

If `n` $= 1$ *(i.e.,* `v0` $= 1$*), neither* `color(2)` *nor* `color(3)` *are associated with a distribution. Thus,* `rv(color(2),v2)` *fails, and both* `not_red` *and* `not_red_either` *therefore fail as well, independently of the values of the comparison literals. In contrast, under the procedural semantics of Nitti et al. [2016],* `color(2)=:=red` *fails in this case,*

[6] Local stratification is a necessary condition for perfect models semantics, and a sufficient one for well-founded semantics. On this class of programs, both semantics agree [Van Gelder et al., 1991]

and `not_red` *thus succeeds. Similarly,* `color(2)=\=red` *fails, and* `not_red_either` *thus fails. Both views agree for* $n > 1$.

This example again illustrates that DC-ProbLog's semantics clearly follows the spirit of the distribution semantics of defining a distribution over interpretations of basic facts (comparison atoms in this case) independently of the logic program rules. We note that the expressive power of logic programs allows the programmer to explicitly model the procedural view of "failure through undefined variable" in the program if desired, as illustrated in the following example.

Example E.2. *The following DC-ProbLog program is equivalent to the procedural interpretation of the program in Example E.1:*

```
1 n ~ uniform([1,2,3]).
2 color(1) ~ uniform([red,green,blue]) :- 1=<n .
3 color(2) ~ uniform([red,green,blue]) :- 2=<n .
4 color(3) ~ uniform([red,green,blue]) :- 3=<n .
5 not_red :- not color(2)=:=red.
6 not_red :- not 2=<n.
7 not_red_either :- 2=<n, color(2)=\=red.
```

We explicitly model that `not_red` *is true if either* `color(2)` *can be interpreted and* `color(2)=:=red` *is false (line 5, which is how DC-ProbLog interprets the first clause), or* `color(2)` *cannot be interpreted (line 6, negating the body of the DC in line 3). Similarly,* `not_red_either` *is true if and only if* `color(2)` *can be resolved and* `color(2)=\=red` *is true (line 7, repeating the body of the DC in line 3).*

E.2. Program validity

To define valid programs, Gutmann et al. [2011] impose acyclicity criteria based on the structure of the clauses in the program, whereas we use the ancestor relation between random variables in DC-ProbLog. This means that DC-ProbLog accepts certain cycles in the logic program structure that are rejected by Distributional Clauses, as illustrated in the following example.

Example E.3. *We model a scenario where a property of a node in a network is either initiated locally with probability* 0.1*, or propagated from a neighboring node that has the property with probability* 0.3*. We consider a two node network with directed edges from each of the nodes to the other one, and directly ground the program for this situation.*

```
local(n1) ~ flip(0.1).
local(n2) ~ flip(0.1).
transmit(n1,n2) ~ flip(0.3) :- active(n1).
transmit(n2,n1) ~ flip(0.3) :- active(n2).
active(n1) :- local(n1)=:=1.
active(n2) :- local(n2)=:=1.
active(n1) :- transmit(n2,n1)=:=1.
active(n2) :- transmit(n1,n2)=:=1.
```

This program is not distribution-stratified based on [Gutmann et al., 2011], where in order to avoid cyclic probabilistic dependencies 1) DC heads have to be of strictly higher rank than any of their body atoms, 2) heads of regular clauses have to have at least the same rank as each body atom, and 3) atoms involving random terms have to have at least the same rank as the head of the DC introducing the random term. This is impossible with our program due to the cyclic dependency between `active`*-atoms and* `transmit`*-random terms. The DC-ProbLog semantics, in contrast, is clearly specified through the mapping:*

```
x1 ~ flip(0.1).
x2 ~ flip(0.1).
x3 ~ flip(0.3).
x4 ~ flip(0.3).
rv(local(n1),x1).
rv(local(n2),x2).
rv(transmit(n1,n2),x3) :- active(n1).
rv(transmit(n2,n1),x4) :- active(n2).
active(n1) :- rv(local(n1),x1), x1=:=1.
active(n2) :- rv(local(n2),x2), x2=:=1.
active(n1) :- rv(transmit(n2,n1),x4), x4=:=1.
active(n2) :- rv(transmit(n1,n2),x3), x3=:=1.
```

We have four independent random variables, and a definite clause program whose meaning is well-defined despite of the cyclic dependencies between derived atoms. We can equivalently rewrite the logic program part to avoid deterministic auxiliaries:

```
x1 ~ flip(0.1).
x2 ~ flip(0.1).
x3 ~ flip(0.3).
x4 ~ flip(0.3).
active(n1) :- x1=:=1.
active(n2) :- x2=:=1.
active(n1) :- active(n2), x4=:=1.
active(n2) :- active(n1), x3=:=1.
```

Furthermore, DC-ProbLog agrees with the ProbLog formulation of the original program, i.e.,

```
0.1::local_cause(n1).
0.1::local_cause(n2).
0.3::transmit_cause(n1,n2) :- active(n1).
0.3::transmit_cause(n2,n1) :- active(n2).
active(n1) :- local_cause(n1).
active(n2) :- local_cause(n2).
active(n1) :- transmit_cause(n2,n1).
active(n2) :- transmit_cause(n1,n2).
```

The AD-free program is

```
v1 ~ flip(0.1).
local_cause(n1) :- v1=:=1.
v2 ~ flip(0.1).
local_cause(n2) :- v2=:=1.
v3 ~ flip(0.3).
transmit_cause(n1,n2) :- v3=:=1, active(n1).
v4 ~ finite(0.3).
transmit_cause(n2,n1) :- v4=:=1, active(n2).
active(n1) :- local_cause(n1).
active(n2) :- local_cause(n2).
active(n1) :- transmit_cause(n2,n1).
active(n2) :- transmit_cause(n1,n2).
```

As this already is a DF-PLP program, we can skip the further rewrites. While the definite clauses are factored differently compared to the earlier variants, their meaning is the same.

F. Proofs of Theorems and Propositions in Section 6 and Section 7

F.1. Proof of Theorem 6.2

Theorem 6.2 (Label Equivalence)**.** *Let $\mathcal{P}$ be a DC-ProbLog program and let $\mathcal{P}_g$ be the relevant ground program for $\mathcal{P}$ with respect to a query μ and the evidence $\mathcal{E} = e$ obtained by first grounding out logical variables and subsequently applying transformation rules from Section 4. The programs $\mathcal{P}$ and $\mathcal{P}_g$ specify the same probability:*

$$P_{\mathcal{P}}(\mu = \top \mid \mathcal{E} = e) = P_{\mathcal{P}_g}(\mu = \top \mid \mathcal{E} = e) \tag{6.1}$$

Proof. The semantics of $\mathcal{P}$ is given by the ground program that is obtained by first grounding $\mathcal{P}$ with respect to its Herbrand base and reducing it to a DF-PLP program as specified in Section 4. The resulting program consists of distributional facts and ground normal clauses only, and includes clauses defining rv-atoms as well as calls to those atoms in clause bodies. However, as the definitions of these atoms are acyclic, and each ground instance is defined by a single rule (with the body of the DC that introduced the new random variable), we can eliminate all references to such atoms by recursively applying the well-known *unfolding* transformation, which replaces atoms in clause bodies by their definition. The result is an equivalent ground program using only predicates from $\mathcal{P}$, but where rule bodies have been expanded with the contexts of the random variables they interpret. We know from Theorem 1 in [Fierens et al., 2015] that for given query and evidence, it is sufficient to use the part of this logic program that is encountered during backward chaining from those atoms. We note that in our case, this also includes the distributional facts providing the distributions for relevant random variables, i.e., random variables in relevant comparison atoms as well as their ancestors. □

F.2. Proof of Theorem 6.9

Theorem 6.9 (Label Equivalence)**.** *Let $\mathcal{P}_g$ be the relevant ground program for a DC-ProbLog program $\mathcal{P}$ with respect to a query μ and the evidence $\mathcal{E} = e$. Let ϕ_g denote the propositional formula derived from $\mathcal{P}_g$ and let α be the labeling function as defined in Definition 6.5. We then have* **label equivalence***, i.e.*

$$\forall \varphi \in ENUM(\phi_g) : \mathop{\mathbb{E}}_{\mathcal{V}\sim\mathcal{P}_g}[\alpha(\varphi)] = P_{\mathcal{P}_g}(\varphi) \tag{6.5}$$

In other words, for all models φ of ϕ_g, the expected value ($\mathbb{E}[\cdot]$) of the label of φ is equal to the probability of φ according to the probability measure of relevant ground program $\mathcal{P}_g$.

Proof. The probability of a model φ of the relevant ground program $\mathcal{P}_g$ is, according to the distribution semantics (cf. Appendix C.2), given by:

$$P_{\mathcal{P}_g}(\varphi) = \int \mathbf{1}_{[\mu_1=b_1\wedge\ldots\wedge\mu_n=b_n]}(\omega(\mathcal{V}))\, dP_{\mathcal{V}} \tag{F.1}$$

where the μ_i are the comparison atoms that appear (positively or negatively) in $\mathcal{P}_g$ and the b_i the truth values these atoms take in φ. We can manipulate the probability into:

$$\begin{aligned}
P_{\mathcal{P}_g} &= \int \left(\prod_{i=1}^{n} \mathbf{1}_{[\mu_i=b_i]}(\omega(\mathcal{V})) \right) dP_{\mathcal{V}} && \text{(F.2)}\\
&= \int \left(\prod_{i:b_i=\bot} \mathbf{1}_{[\mu_i=b_i]}(\omega(\mathcal{V})) \right)\left(\prod_{i:b_i=\top} \mathbf{1}_{[\mu_i=b_i]}(\omega(\mathcal{V})) \right) dP_{\mathcal{V}} && \text{(F.3)}\\
&= \int \left(\prod_{i:b_i=\bot} [\![\neg c_i(vars(\mu_i))]\!] \right)\left(\prod_{i:b_i=\top} [\![c_i(vars(\mu_i))]\!] \right) dP_{\mathcal{V}} && \text{(F.4)}
\end{aligned}$$

Turning our attention now to the expected value of $\alpha(\varphi)$ we have:

$$\mathop{\mathbb{E}}_{\mathcal{V}\sim\mathcal{P}_g}[\alpha(\varphi)] = \int \alpha\left(\bigwedge_{\ell_i\in\varphi} \ell_i \right) dP_{\mathcal{V}} = \int \left(\prod_{\ell_i\in\varphi} \alpha\,(\ell_i) \right) dP_{\mathcal{V}} \tag{F.5}$$

The literals $\ell_i \in \varphi$ fall into four groups: atoms whose predicate is a comparison and that are true in φ (denoted by $CA^+(\varphi)$), non-comparison atoms that are true in φ (denoted $NA^+(\varphi)$), and similarly the atoms that are false in φ (denoted by $CA^-(\varphi)$ and $NA^-(\varphi)$). This yields:

$$\begin{aligned}
&\mathop{\mathbb{E}}_{\mathcal{V}\sim\mathcal{P}_g}[\alpha(\varphi)] && \text{(F.6)}\\
&= \int \left(\prod_{\ell_i\in CA^+(\varphi)} \alpha(\ell_i) \right)\left(\prod_{\ell_i\in CA^-(\varphi)} \alpha(\neg\ell_i) \right)\left(\prod_{\ell_i\in NA^+(\varphi)} \alpha(\ell_i) \right)\left(\prod_{\ell_i\in NA^-(\varphi)} \alpha(\neg\ell_i) \right) dP_{\mathcal{V}}
\end{aligned}$$

Plugging in the definition of the labeling function the last two products reduce to 1 and we obtain for the remaining expression:

$$\mathop{\mathbb{E}}_{\mathcal{V}\sim\mathcal{P}_g}[\alpha(\varphi)] = \int \left(\prod_{i:\ell_i\in CA^+(\varphi)} [\![c_i(vars(\ell_i))]\!] \right)\left(\prod_{i:\ell_i\in CA^-(\varphi)} [\![\neg c_i(vars(\ell_i))]\!] \right) dP_{\mathcal{V}} \tag{F.7}$$

Identifying now the set $\{i : \ell_i \in CA^+(\varphi)\}$ with the set $\{i : \mu_i = \top\}$ and the set $\{i : \ell_i \in CA^-(\varphi)\}$ with the set $\{i : \mu_i = \bot\}$ proves the theorem, as this equates Equation F.4 and Equation F.7 □

F.3. Proof of Proposition 7.1

Proposition 7.1 (Monte Carlo Approximation of a Conditional Query)**.** *Let the set*

$$\mathcal{S} = \left\{ \left(s_1^{(1)}, \dots, s_M^{(1)} \right), \dots, \left(s_1^{(|\mathcal{S}|)}, \dots, s_M^{(|\mathcal{S}|)} \right) \right\} \tag{7.2}$$

denote $|\mathcal{S}|$ i.i.d. samples for each random variable in $\mathcal{P}_g$. A conditional probability query to a DC-ProbLog program $\mathcal{P}$ can be approximated as:

$$P_{\mathcal{P}}(\mu = q \mid \mathcal{E} = e) \approx \frac{\sum_{i=1}^{|\mathcal{S}|} \sum_{\varphi \in ENUM(\phi \wedge \phi_q)} \alpha^{(i)}(\varphi)}{\sum_{i=1}^{|\mathcal{S}|} \sum_{\varphi \in ENUM(\phi)} \alpha^{(i)}(\varphi)}, \qquad N < \infty \tag{7.3}$$

The index (i) on $\alpha^{(i)}(\varphi)$ indicates that the label of φ is evaluated at the i-th ordered set of samples $\left(s_1^{(i)}, \dots, s_M^{(i)} \right)$.

Proof. First we write the conditional probability as a ratio of expected values invoking Theorem 6.10, on which we then use Defintion 6.7:

$$P_{\mathcal{P}}(\mu = q \mid \mathcal{E} = e) = \frac{\mathbb{E}_{\mathcal{V} \sim \mathcal{P}_g}[\alpha(\phi \wedge \phi_q)]}{\mathbb{E}_{\mathcal{V} \sim \mathcal{P}_g}[\alpha(\phi)]} \tag{F.8}$$

$$= \frac{\mathbb{E}_{\mathcal{V} \sim \mathcal{P}_g}\left[\sum_{\varphi \in ENUM(\phi \wedge \phi_q)} \prod_{\ell \in \varphi} \alpha(\ell) \right]}{\mathbb{E}_{\mathcal{V} \sim \mathcal{P}_g}\left[\sum_{\varphi \in ENUM(\phi)} \prod_{\ell \in \varphi} \alpha(\ell) \right]} \tag{F.9}$$

$$= \frac{\mathbb{E}_{\mathcal{V} \sim \mathcal{P}_g}\left[\sum_{\varphi \in ENUM(\phi \wedge \phi_q)} \alpha(\varphi) \right]}{\mathbb{E}_{\mathcal{V} \sim \mathcal{P}_g}\left[\sum_{\varphi \in ENUM(\phi)} \alpha(\varphi) \right]} \tag{F.10}$$

We can now express the conditional probability in terms of the sampled values $\mathcal{S}$:

$$P_{\mathcal{P}}(\mu = q \mid \mathcal{E} = e) = \frac{\lim_{\to \infty} {}^{1}\!/\!_{|\mathcal{S}|} \sum_{i=1}^{|\mathcal{S}|} \sum_{\varphi \in ENUM(\phi \wedge \phi q)} \alpha^{(i)}(\varphi)}{\lim_{N \to \infty} {}^{1}\!/\!_{|\mathcal{S}|} \sum_{i=1}^{|\mathcal{S}|} \sum_{\varphi \in ENUM(\phi)} \alpha^{(i)}(\varphi)} \tag{F.11}$$

$$\approx \frac{\sum_{i=1}^{|\mathcal{S}|} \sum_{\varphi \in ENUM(\phi \wedge \phi_q)} \alpha^{(i)}(\varphi)}{\sum_{i=1}^{|\mathcal{S}|} \sum_{\varphi \in ENUM(\phi)} \alpha^{(i)}(\varphi)}, \qquad |\mathcal{S}| < \infty \tag{F.12}$$

□

F.4. Proof of Proposition 7.13

Proposition 7.13 (Consistency of IALW)**.** *Infinitesimal algebraic likelihood weighting is consistent, that is, the approximate equality in Equation 7.12 is almost surely an equality for $|\mathcal{S}| \to \infty$.*

Proof. First we manipulate the expected value on the left hand side of Equation 7.12:

$$\mathbb{E}\left[\sum_{\varphi\in ENUM(\phi)}\prod_{\ell\in\varphi}\alpha(\ell)\middle|\mathcal{S}\right] \tag{F.13}$$

$$= \lim_{|\mathcal{S}|\to\infty}\sum_{i=1}^{|\mathcal{S}|}\sum_{\varphi\in ENUM(\phi)}\prod_{\ell\in\varphi}\alpha^{(i)}(\ell) \tag{F.14}$$

$$= \lim_{|\mathcal{S}|\to\infty}\sum_{i=1}^{|\mathcal{S}|}\sum_{\varphi\in ENUM(\phi)}\left(\prod_{\ell\in\varphi\setminus DI(\varphi)}\alpha^{(i)}(\ell)\prod_{\ell\in DI(\varphi)}\alpha^{(i)}(\ell)\right) \tag{F.15}$$

As the samples are ancestral samples, they satisfy by construction the delta invervals appearing in the second product. This means that $\prod_{\ell\in DI(\varphi)}\alpha^{(i)}(\ell) = 1$ and that we can write the expected value in function of non delta interval atoms only:

$$\mathbb{E}\left[\sum_{\varphi\in ENUM(\phi)}\prod_{\ell\in\varphi}\alpha(\ell)\middle|\mathcal{S}\right] = \mathbb{E}\left[\underbrace{\sum_{\varphi\in ENUM(\phi)}\prod_{\ell\in\varphi\setminus DI(\varphi)}\alpha(\ell)}_{:=f(\phi)}\middle|\mathcal{S}\right] \tag{F.16}$$

Let us now manipulate the expression in the numerator on the right hand side of Equation 7.12:

$$\bigoplus_{i=1}^{|\mathcal{S}|}\bigoplus_{\varphi\in ENUM(\phi)}\bigotimes_{\ell\in\varphi}\alpha_{IALW}^{(i)}(\ell) \tag{F.17}$$

$$= \bigoplus_{i=1}^{|\mathcal{S}|}\bigoplus_{\varphi\in ENUM(\phi)}\underbrace{\left(\bigotimes_{\ell\in\varphi\setminus DI(\varphi)}\alpha_{IALW}^{(i)}(\ell)\right)}_{\left(r_\varphi^{(i)},0\right)}\otimes\underbrace{\left(\bigotimes_{\ell\in DI(\varphi)}\alpha_{IALW}^{(i)}(\ell)\right)}_{\left(t_\varphi^{(i)},m_\varphi^{(i)}\right)} \tag{F.18}$$

The expressions $\left(r_\varphi^{(i)},0\right)$ and $\left(t_\varphi^{(i)},m_\varphi^{(i)}\right)$ denote infinitesimal numbers. Note how only the latter of the two picks up a non-zero second part.

From the definition of the addition of two infinitesimal numbers we can see that only those infinitesimal numbers with the smallest integer in the second part *survive* the addition. This also means that in Equation F.18 only those terms that have the smallest integer in their second part among all terms will contribute. We denote this smallest integer by:

$$m^* = \min_{\substack{i\in\{1,\ldots,|\mathcal{S}|\}\\ \varphi\in ENUM(\phi)}} m_\varphi^{(i)} \tag{F.19}$$

We rewrite Equation F.18 in function of m^*:

$$\bigoplus_{i=1}^{|\mathcal{S}|} \bigoplus_{\varphi \in ENUM(\phi)} \left([\![m_\varphi^{(i)} {=} m^*]\!], 0 \right) \otimes \left(r_\varphi^{(i)}, 0 \right) \otimes \left(t_\varphi^{(i)}, m^* \right) \tag{F.20}$$

$$= \bigoplus_{i=1}^{|\mathcal{S}|} \bigoplus_{\varphi \in ENUM(\phi)} \left([\![m_\varphi^{(i)} {=} m^*]\!] r_\varphi^{(i)} t_\varphi^{(i)}, 0 \right) \otimes (1, m^*) \tag{F.21}$$

$$= \left(\sum_{i=1}^{|\mathcal{S}|} \sum_{\varphi \in ENUM(\phi)} [\![m_\varphi^{(i)} {=} m^*]\!] r_\varphi^{(i)} t_\varphi^{(i)}, 0 \right) \otimes (1, m^*) \tag{F.22}$$

Similarly we get for the denominator in Equation 7.12:

$$\bigoplus_{i=1}^{|\mathcal{S}|} \bigoplus_{\varphi \in ENUM(\phi)} \bigotimes_{\ell \in DI(\varphi)} \alpha_{IALW}^{(i)}(\ell) \tag{F.23}$$

$$= \left(\sum_{i=1}^{|\mathcal{S}|} \sum_{\varphi \in ENUM(\phi)} [\![m_\varphi^{(i)} {=} m^*]\!] t_\varphi^{(i)}, 0 \right) \otimes (1, m^*) \tag{F.24}$$

We can now plug Equation F.16, Equation F.22 and Equation F.24 back into Equation 7.12 and obtain:

$$(\mathbb{E}[f(\phi)|\mathcal{S}], 0) = \left(\frac{\sum_{i=1}^{|\mathcal{S}|} \sum_{\varphi \in ENUM(\phi)} [\![m_\varphi^{(i)} {=} m^*]\!] r_\varphi^{(i)} t_\varphi^{(i)}}{\sum_{i=1}^{|\mathcal{S}|} \sum_{\varphi \in ENUM(\phi)} [\![m_\varphi^{(i)} {=} m^*]\!] t_\varphi^{(i)}}, 0 \right) \quad (|\mathcal{S}| \to \infty) \tag{F.25}$$

$$\Leftrightarrow \mathbb{E}[f(\phi)|\mathcal{S}] = \frac{\sum_{i=1}^{|\mathcal{S}|} \sum_{\varphi \in ENUM(\phi)} [\![m_\varphi^{(i)} {=} m^*]\!] r_\varphi^{(i)} t_\varphi^{(i)}}{\sum_{i=1}^{|\mathcal{S}|} \sum_{\varphi \in ENUM(\phi)} [\![m_\varphi^{(i)} {=} m^*]\!] t_\varphi^{(i)}} \tag{F.26}$$

We realize that $r_\varphi^{(i)}$ is actually $f(\phi)$ evaluated at the i-th sample at the instantiation φ and evoke [Wu et al., 2018, Theorem 4.1] to prove Equation F.26, which also finishes this proof. $\square$

F.5. Proof of Proposition 7.14

Proposition 7.14. *A conditional probability query to a DC-ProbLog program $\mathcal{P}$ can be approximated as:*

$$P_{\mathcal{P}}(\mu = q|\mathcal{E} = e) \approx \frac{\bigoplus_{i=1}^{|\mathcal{S}|} \bigoplus_{\varphi \in ENUM(\phi \wedge \phi_q)} \bigotimes_{\ell \in \varphi} \alpha_{IALW}^{(i)}(\ell)}{\bigoplus_{i=1}^{|\mathcal{S}|} \bigoplus_{\varphi \in ENUM(\phi)} \bigotimes_{\ell \in \varphi} \alpha_{IALW}^{(i)}(\ell)} \tag{7.13}$$

Proof. We start the proof by invoking Theorem 6.10, which expresses the conditional probability as a ratio of expectations. In the numerator and the denominator we then write the label of the propositional logic formulas as the sum of the labels of the respective possible worlds.

$$P_{\mathcal{P}}(\mu = q | \mathcal{E} = e) = \frac{\mathbb{E}_{\mathcal{V}\sim\mathcal{P}_g}[\alpha(\phi \wedge \phi_q)]}{\mathbb{E}_{\mathcal{V}\sim\mathcal{P}_g}[\alpha(\phi)]} \tag{F.27}$$

$$= \frac{\mathbb{E}_{\mathcal{V}\sim\mathcal{P}_g}\left[\sum_{\varphi\in ENUM(\phi\wedge\phi_{qe})} \alpha(\varphi)\right]}{\mathbb{E}_{\mathcal{V}\sim\mathcal{P}_g}\left[\sum_{\varphi\in ENUM(\phi)} \alpha(\varphi)\right]} \tag{F.28}$$

Next, we approximate the expectation using a set of ancestral samples $\mathcal{S}$, followed by pulling out the query from the summation index in the numerator:

$$\frac{\mathbb{E}_{\mathcal{V}\sim\mathcal{P}_g}\left[\sum_{\varphi\in ENUM(\phi\wedge\phi_q)} \alpha(\varphi)\right]}{\mathbb{E}_{\mathcal{V}\sim\mathcal{P}_g}\left[\sum_{\varphi\in ENUM(\phi)} \alpha(\varphi)\right]} \tag{F.29}$$

$$\approx \frac{\mathbb{E}\left[\sum_{\varphi\in ENUM(\phi\wedge\phi_q)} \alpha(\varphi)|\mathcal{S}\right]}{\mathbb{E}\left[\sum_{\varphi\in ENUM(\phi)} \alpha(\varphi)|\mathcal{S}\right]} \tag{F.30}$$

$$\approx \frac{\mathbb{E}\left[\sum_{\varphi\in ENUM(\phi)} [\![\varphi \models \phi_q]\!] \alpha(\varphi)|\mathcal{S}\right]}{\mathbb{E}\left[\sum_{\varphi\in ENUM(\phi)} \alpha(\varphi)|\mathcal{S}\right]} \tag{F.31}$$

We now rewrite the fraction of two real numbers in Equation F.31 as the fraction of two infinitesimal numbers and plug in the definition of the infinitesimal algebraic likelihood weight (cf. Definition 7.12):

$$\frac{\mathbb{E}\left[\sum_{\varphi\in ENUM(\phi)} [\![\varphi \models \phi_q]\!] \alpha(\varphi)|\mathcal{S}\right]}{\mathbb{E}\left[\sum_{\varphi\in ENUM(\phi)} \alpha(\varphi)|\mathcal{S}\right]} \tag{F.32}$$

$$= \frac{\left(\mathbb{E}\left[\sum_{\varphi\in ENUM(\phi)} [\![\varphi \models \phi_q]\!] \alpha(\varphi)|\mathcal{S}\right], 0\right)}{\left(\mathbb{E}\left[\sum_{\varphi\in ENUM(\phi)} \alpha(\varphi)|\mathcal{S}\right], 0\right)} \tag{F.33}$$

$$\approx \frac{\bigoplus_{i=1}^{|\mathcal{S}|} \bigoplus_{\varphi\in ENUM(\phi)} [\![\varphi \models \phi_q]\!] \bigotimes_{\ell\in\varphi} \alpha_{IALW}^{(i)}(\ell)}{\cancel{\bigoplus_{i=1}^{|\mathcal{S}|} \bigoplus_{\varphi\in ENUM(\phi)} \bigotimes_{\ell\in DI(\varphi)} \alpha_{IALW}^{(i)}(\ell)}} \otimes \frac{\cancel{\bigoplus_{i=1}^{|\mathcal{S}|} \bigoplus_{\varphi\in ENUM(\phi)} \bigotimes_{\ell\in DI(\varphi)} \alpha_{IALW}^{(i)}(\ell)}}{\bigoplus_{i=1}^{|\mathcal{S}|} \bigoplus_{\varphi\in ENUM(\phi)} \bigotimes_{\ell\in\varphi} \alpha_{IALW}^{(i)}(\ell)} \tag{F.34}$$

In the last line the first factor corresponds to the numerator of the previous equation and the second factor corresponds to the reciprocal of the denominator. Note that the consistency of the infinitesimal algebraic likelihood weight of the numerator (first factor) is guaranteed by defining a new labeling function $\alpha^q(\varphi) := [\![\varphi \models \phi_q]\!]\alpha(\varphi)$ and evoking Proposition 7.13 with α^q.

Finally, we push the expression $[\![\varphi \models \phi_q]\!]$ in the numerator back into the index of the summation ($\oplus$), which yields the following expression:

$$\frac{\bigoplus_{i=1}^{|\mathcal{S}|} \bigoplus_{\varphi\in ENUM(\phi\wedge\phi_q)} \bigotimes_{\ell\in\varphi} \alpha_{ALWI}^{(i)}(\ell)}{\bigoplus_{i=1}^{|\mathcal{S}|} \bigoplus_{\varphi\in ENUM(\phi)} \bigotimes_{\ell\in\varphi} \alpha_{IALW}^{(i)}(\ell)} \tag{F.35}$$

which proves the proposition. □

F.6. Proof of Proposition 7.18

Proposition 7.18 (ALW on d-DNNF)**.** *We are given the propositional formulas ϕ and ϕ_q and a set $\mathcal{S}$ of ancestral samples, we can use Algorithm 7.19 to compute the conditional probability $P_{\mathcal{P}}(\mu = q|\mathcal{E} = e)$.*

Proof. Algorithm 7.19 first compiles both propositional formulas into equivalent d-DNNF representations, cf. Lines 2 and 3. In Lines 4 and 5 it then computes the (unnormalized) infinitesimal algebraic likelihood weight for both formulas by calling Algorithm 7.20. In other words, we compute the numerator and denominator in Equation F.35. We observe that Algorithm 7.20 evaluates a given d-DNNF formula for each conditioned topological sample using the `Eval` function, which evaluates a d-DDNF formula given a labeling function, cf. Algorithm 7.21 [Kimmig et al., 2017]. The correctness of Algorithm 7.19 now hinges on the correctness of the `Eval` function, which was proven by Kimmig et al. [2017] for the evaluation of a d-DNNF formula using a semiring and labeling function pair that adheres to the properties described in Lemmas 7.15 to 7.17. Effectively, Algorithm 7.20 correctly computes the algebraic model count for each ancestral sample, adds up the results, and returns the unnormalized algebraic model count to Algorithm 7.19. Line 6 finally return the ratio of the two unnormalized algebraic likelihood weights, which corresponds to the conditional probability $P_{\mathcal{P}}(\mu = q|\mathcal{E} = e)$, as proven in Equations F.27 to F.35. □