Syntactic Requirements for Well-defined Hybrid Probabilistic Logic Programs

Damiano Azzolini

University of Ferrara
Dipartimento di Ingegneria

damiano.azzolini@unife.it

Fabrizio Riguzzi

University of Ferrara
Dipartimento di Matematica e Informatica

fabrizio.riguzzi@unife.it

Hybrid probabilistic logic programs can represent several scenarios thanks to the expressivity of Logic Programming extended with facts representing discrete and continuous distributions. The semantics for this type of programs is crucial since it ensures that a probability can be assigned to every query. Here, following one recent semantics proposal, we illustrate a concrete syntax, and we analyse the syntactic requirements needed to preserve the well-definedness.

1 Introduction

The power and expressivity of Probabilistic Logic Programming (PLP) [8, 18] have been utilized to represent many real world situations [2, 9, 14]. Usually, probabilistic logic programs involve only discrete random variables with Bernoulli or Categorical distributions. Numerous solutions emerged to also handle continuous distributions [10, 12, 25], increasing the expressiveness of PLP and giving birth to *hybrid* probabilistic logic programs, that is, programs that include discrete and continuous random variables. Inference in this type of programs is hard since it combines the complexity of the grounding computation with the intractability of a distribution defined by a mixture of random variables.

Usually, inference in general hybrid probabilistic logic programs (i.e., without imposing restrictions on the type of distributions allowed) is done by leveraging knowledge compilation and using external solvers [25] or by sampling [4, 16].

The semantics of hybrid programs has been proved well-defined in [3], but the authors neither provided an explicit syntax, nor introduced explicit syntactic requirements. In this paper, we close this gap by introducing an explicit syntax and the syntactic requirements needed to preserve the well-definedness of the semantics of hybrid probabilistic logic programs.

The paper is structured as follows: in Section 2 we introduce the semantics and the syntax for programs with only discrete random variables with both finite and infinite number of explanations. The semantics and syntax for hybrid programs is introduced in Section 3, and the syntactic requirements are analysed in Section 4. Section 5 concludes the paper. All the Sections are accompanied by several examples, to make the concepts clearer.

2 Probabilistic Logic Programming

Probabilistic Logic Programming (PLP) allows the construction of complex yet interpretable probability models through the usage of Logic Programming. Several probabilistic logic languages based on the Distribution Semantics (DS) [21] have been proposed during the years, such as PRISM [21], and ProbLog [9]. Basically, they allow the definition of probabilistic facts, i.e., facts that can be true or false with a certain probability.

Following the ProbLog syntax [9], each probabilistic fact has the form $p :: a$ where $p \in\,]0,1]$, meaning that a is true with probability p and false with probability $1-p$. If $p = 1$, the fact is deterministic. An *atomic choice* indicates whether a grounding $a\theta$ of a probabilistic fact $p :: a$ is selected or not, and it is represented with the triple (a, θ, k). $k \in \{0,1\}$, where $k = 1$ means that the fact is selected and $k = 0$ that it is not. If a set of atomic choices does not contain two atomic choices in which the fact is both selected and not, it is termed *consistent*. A consistent set κ of atomic choices is called *composite choice* and its probability can be computed as:

$$P(\kappa) = \prod_{(a_i,\theta,1)\in\kappa} p_i \cdot \prod_{(a_i,\theta,0)\in\kappa} (1 - p_i)$$

A *total* composite choice contains one atomic choice for every grounding of every probabilistic fact. A *world* is a logic program identified by a total composite choice, and it is created by selecting the atoms corresponding to each atomic choice with $k = 1$. Its probability is given by the probability of the total composite choice. Here we consider only programs where every world has a two-valued Well-Founded Model (WFM). A semantics for programs without a two-valued WFM [22] based on credal sets is discussed in [7]. If a program does not contain function symbols, the set of groundings for every probabilistic fact is finite, and so the set of worlds for a program is finite.

Example 1 (Card) *Consider the following toy example that models a deck of three cards composed of ace of spades, ace of clubs, and ace of hearts. The three cards can be drawn with equal probability. This simple scenario can be modelled using the following program:*

```
1/3 :: spades(X).
1/2 :: clubs(X).
pick(0,spades) :- spades(0).
pick(0,clubs) :- \+ spades(0), clubs(0).
pick(0,hearts) :- \+ spades(0), \+ clubs(0).
```

We use only two random variables since the third (hearts) can be represented with both random variables negated (see 5th clause). Furthermore, note that the probability of `clubs(X)` is set to $1/2$ to obtain $1/3$ for every possible choice: $P(pick(0,spades)) = 1/3$, $P(pick(0,clubs)) = 2/3 \cdot 1/2 = 1/3$, and $P(pick(0,hearts)) = 2/3 \cdot 1/2 = 1/3$. We keep the variable X for the first two facts, even if it may be considered a singleton variable, to better illustrate the next concepts.

Example 1 illustrates a program without function symbols. If a program has at least one variable, one constant, and one function symbol, its grounding is infinite, so the previous definition must be extended [15].

The set of worlds ω_κ compatible with a composite choice κ is defined as $\omega_\kappa = \{w_\sigma \in W_P \mid \kappa \subseteq \sigma\}$, where W_P is the set of worlds for the whole probabilistic logic program, σ is a total composite choice, and w_σ is the world identified by the total choice. If the program does not have function symbols, the probability of a composite choice is equal to the sum of the probabilities of the worlds in ω_κ. If the program has function symbols, ω_κ may be uncountable and the probability of each world is 0 (infinite product of values $\in\,]0,1[$). However, the probability of a composite choice can still be computed. The set of worlds *compatible* with a set of composite choices K is given by the union of the worlds identified by each composite choice in the set. Two composite choices are *incompatible* if their union is not consistent. A set of composite choices is *pairwise incompatible* if every pair of different composite choices

is incompatible. $P(K)$ is still well-defined for programs with function symbols and can be computed by constructing pairwise incompatible equivalent sets. A composite choice κ is an *explanation* for a query q (a conjunction of ground atoms) if, for all worlds $w \in \omega_\kappa$, $w \models q$. Furthermore, a set of composite choices is *covering* with respect to a query q if $\forall w$ in which q is true, $w \in \omega_\kappa$. We can also define as explanation a set of worlds and define the covering property for sets of worlds. The probability of a query q is then computed by defining a probability measure over the set of worlds identified by countable sets of countable composite choices: $\mu_P(\omega_K) = \lim_{n\to\infty} \mu(K'_n)$, where $K'_n = \{\kappa'_1, \ldots, \kappa'_n\}$ is a pairwise in-compatible set of composite choices equivalent to $K_n = \{\kappa_1, \ldots, \kappa_n\}$, $K = \{\kappa_1, \kappa_2, \ldots\}$ is a covering set of explanations for q and $\mu_P(K_n) = \sum_{i=1}^{n} P(\kappa'_i)$. Programs with function symbols have a well-defined semantics [17].

If a probabilistic logic program is range restricted, i.e., every variable in the head appears also in a positive literal in the body, the answers of a query are always ground instantiations of it [13, 20], and probabilities are always assigned to ground atoms. However, in some cases, probabilistic facts may not be range restricted, and can contain variables (see, for instance, the probabilistic facts spades/1 and clubs/1 in Example 1). In this situation, queries' answers are still ground instantiations and the program can be well-defined provided that the variables in probabilistic facts are ground when the fact is called: they can be directly bound to a constant (as in Example 1, where X is bound to 0), or they must appear in a previous literal in the body.

Theorem 1 (Ground queries with well-defined probability) *Given a Probabilistic Logic Program P and a query q, if P is range restricted and all the variables in probabilistic facts appearing in the body of clauses of P are present in a previous positive literal, the answers to q will be ground instantiation of it, with an associated well-defined probability.*

Before illustrating the proof, we introduce some basic concepts following [13]. Given two atoms, a and b, a substitution θ (a replacement of variables with terms) is a unifier for a and b if $a\theta = b\theta$. A substitution is a *grounding* if all the involved terms are ground. Furthermore, a unifier σ is the most general unifier (mgu) for a and b if, for all unifiers μ of both a and b, exists a substitution ρ such that $(a\sigma)\rho = a\mu$.

The main step of *resolution* is the following: given two clauses A and B with no common variables, $l \in A$ and $\neg l' \in B$, and θ being the mgu of l and l', the *resolvent* of A and B is defined as $((A \setminus \{l\}) \cup (B \setminus \{l'\}))\theta$; for a program P and a query q, resolution is linear when B represents only clauses in P and A is the query or the resolvent of another linear resolution. We suppose that literals in clauses are ordered and are chosen following this order during the resolution. A resolution proof for SLDNF can be represented as a sequence of clauses $\langle q, C_1, C_2, \ldots, C_n \rangle$. If C_n is the empty clause, the proof is a *refutation*. The substitution for the query q is given by $\theta = \theta_1 \theta_2 \ldots \theta_n$, where each θ_i is the substitution correspondent to clause C_i. Moreover, if P is range restricted, θ is a grounding.

Proof 1 *Let us now prove Theorem 1 by induction. In the base case, the SLDNF resolution consists of only one step: the query q unifies with a deterministic fact, since there is not a previous positive literal in the body that will allow the presence of a probabilistic fact. The program is range restricted, θ_1 is computed, and the variables in the query are all grounded. Suppose now that the theorem is true at step n. The current set of substitutions is $\{\theta_1, \theta_2, \ldots, \theta_n\}$. There are two possible cases: if the selected literal of the query matches a deterministic fact, substitution θ_{n+1} grounds the literal, as deterministic facts are already ground. If the selected literal of q matches with a probabilistic fact, this fact cannot be the first of the body of the correspondent clause, by assumption: all the variables appearing in the probabilistic fact are already grounded by preceding substitutions, so the literal is ground when called, and a new*

substitution θ_{n+1} *is computed and added to the list. Eventually, all the variables will be grounded and the answer to the query will be a ground instantiation.*

If variables appear for the first time in probabilistic facts, the query could still eventually be ground, but probabilistic facts will not be ground when called, and thus the semantics will be ill-defined.

Let us now extend Example 1 by introducing new clauses to obtain a program with an infinite covering set of explanations.

Example 2 (Card with an infinite number of explanations) *Consider a game of card where a player needs to pick one out of three possible cards: ace of spades, ace of clubs or ace of hearts. The game stops when the player picks the ace of hearts. This can be modelled by adding the following clauses to Example 1.*

```
pick(s(X),spades):- \+ pick(X,hearts), spades(s(X)).
pick(s(X),clubs):- \+ pick(X,hearts), \+ spades(s(X)), clubs(s(X)).
pick(s(X),hearts):- \+ pick(X,hearts), \+ spades(s(X)), \+ clubs(s(X)).

at_least_once_spades :- pick(_,spades).
never_spades :- \+ at_least_once_spades.
```

We can ask for the probability that the player picks at least once spades or that that he/she never picks spades (respectively $P(\texttt{at\_least\_once\_spades})$ *and* $P(\texttt{never\_spades})$*).*

For conciseness, let us replace spades(X) *with* f_1 *and* clubs(X) *with* f_2*. Consider query* at_least_once_spades*: it has the pairwise incompatible covering set of explanations* $K = \{\kappa_0, \kappa_1, \ldots\}$ *with*

$$\kappa_0 = \{(f_1, \{X/0\}, 1)\}$$
$$\kappa_1 = \{(f_1, \{X/0\}, 0), (f_2, \{X/0\}, 1), (f_1, \{X/s(0)\}, 1)\}$$
$$\cdots$$
$$\kappa_i = \{(f_1, \{X/0\}, 0), (f_2, \{X/0\}, 1), \ldots, (f_1, \{X/s^{i-1}(0)\}, 0),$$
$$(f_2, \{X/s^{i-1}(0)\}, 1), (f_1, \{X/s^i(0)\}, 1)\}$$
$$\cdots$$

In other words, in κ_0 *the player picked spades at round* 0*, in* κ_1 *he/she does not piked spades and picked clubs at round* 0 *and picked spades in round* $s(0)$*, and so on.*

K is countable and infinite and the explanations in it are pairwise incompatible, so the probability of at_least_once_spades *can be computed as*

$$P(at\_least\_once\_spades) = \frac{1}{3} + \frac{1}{3} \cdot \left(\frac{2}{3} \cdot \frac{1}{2}\right) + \frac{1}{3} \cdot \left(\frac{2}{3} \cdot \frac{1}{2}\right)^2 + \ldots$$
$$= \frac{1}{3} + \frac{1}{3} \cdot \left(\frac{1}{3}\right) + \frac{1}{3} \cdot \left(\frac{1}{3}\right)^2 + \ldots$$
$$= \frac{1}{3} \cdot \frac{1}{1 - \frac{1}{3}} = \frac{1}{3} \cdot \frac{3}{2} = \frac{1}{2}$$

since it is the sum of a geometric series. The probability for the query never_spades *can be computed in a similar way. After the computations we get* $P(at\_least\_once\_spades) = 1 - P(never\_spades) = 1/2$*.*

In the next section we introduce the syntax for hybrid probabilistic logic programs, i.e., programs with both discrete and continuous random variables.

3   Hybrid Programs

When a probabilistic logic program contains both discrete and continuous random variables, it is called *hybrid* probabilistic logic program. Here, we also consider constraints involving continuous random variables' values, obtaining probabilistic *constraint* logic programs [3, 12]. In this paper we use hybrid probabilistic logic program and probabilistic constraint logic program interchangeably, since usually in PLP the goal is to compute the probability of a query (and thus we always have constraints to define the range of the continuous random variables), rather than the joint probability density induced by the program.

A probabilistic constraint logic program is composed of a set of rules, a set of Boolean probabilistic facts, and a countable set of continuous random variables. The authors in [3] define $X = \{X_1, X_2, \dots\}$ the countable set of continuous random variables, each one associated with its range (that can be either $\mathbb{R}$ or $\mathbb{R}^n$), and F the set containing discrete probabilistic facts. These discrete facts form a countable set of Boolean random variables $Y = \{Y_1, Y_2, \dots\}$. The sample space of the whole program ($W_{\mathscr{P}}$) is defined as the product of the spaces for continuous (W_X) and discrete (W_Y) random variables, i.e, $W_{\mathscr{P}} = W_X \times W_Y$.

The probabilistic facts associated with every random variable Y_i set to 1, and the grounding of the rules (with the constraints removed) whose constraints are satisfied given a valuation of the continuous random variables, define a ground normal logic program. The probability of a query can be computed by considering a pairwise incompatible covering set of explanations. As before, an explanation for a query of a probabilistic constraint logic program is a set of worlds ω_i such that the query is true in every element of this set.

A concrete syntax for this type of programs is given by `cplint` hybrid programs [18]. In `cplint` hybrid programs, logical variables are partitioned into two disjoint sets: those that can assume terms as values and those that can assume continuous values. Let us call the first *term* variables and the latter *continuous* variables.

Continuous random variables are encoded with probabilistic facts of the form

$$A : Density$$

where A is an atom with a continuous variable *Var* as argument and *Density* is a special atom identifying a probability density on variable *Var*. For example,

`p(X) : gaussian(X,0,1).`

indicates that `X` in atom `p(X)` is a continuous variable that follows a Gaussian distribution with mean 0 and variance 1. Each predicate p/n has a signature that specifies which arguments hold continuous values. Only these arguments can contain continuous variables. Continuous values (and variables) can appear inside a term built on a function symbol f/n. Each function symbol f/n also has a signature that specifies which arguments hold continuous values. Again, only these arguments can contain continuous variables. While discrete random variables are identified by ground atoms (that form a countable set), continuous random variables are identified by predicates and by the ground terms present in atoms with arguments that can hold continuous random variables.

ProbLog probabilistic facts of the form $p :: f$ can also be encoded as $f : p$ for uniformity with Logic Programs with Annotated Disjunctions [24] and CP-Logic [23].

Atoms in clauses and probabilistic facts can have both term and continuous variables. However, we impose the constraint that in every world of the program the values taken by term variables in a ground atom for a predicate p/n that is true in the world uniquely determine the values taken by the continuous variables.

Continuous variables are introduced by probabilistic facts for continuous random variables and by the special predicate =:=/2 that is used to define a new variable based on a formula involving existing continuous variables (see Example 5). Constraints are represented by Prolog comparison predicates. The semantics assigns a probability of being true to any ground atom not having continuous values as arguments. Atoms with continuous values have probability 0, as the probability that a continuous random variable takes a specific value is 0. However, there is one special case when a continuous value is admitted as an argument. This will be discussed in Section 4.

Inference in cplint hybrid programs can be performed using MCINTYRE [1, 4, 19], an algorithm based on Monte Carlo sampling. cplint hybrid programs can also be translated into probabilistic constraint logic programs [12] by removing the continuous variables from the arguments of predicates and by replacing constraints with their probabilistic constraint logic program form.

Consider an extension to Example 2, where we also add a continuous random variable and clauses with constraints.

Example 3 (Card extended with a continuous random variable) *Suppose that the previous game is extended, and the player also needs to spin a wheel. In addition to the previous rules, if the axis of the wheel is between 0 and π degrees (approximated to* 3.14 *for convenience), the game stops. In this example, we have a continuous random variable with uniform distribution that indicates the angle of the wheel, plus a constraint on its value in the clauses for the* pick/2 *predicate:*

```
1/3 :: spades(_).
1/2 :: clubs(_).
angle(_,X) : uniform_dens(X,0,6.28).

pick(0,spades) :- spades(0), angle(0,V), V > 3.14.
pick(0,clubs) :- \+ spades(0), clubs(0), angle(0,V), V > 3.14.
pick(0,hearts) :- \+ spades(0), \+ clubs(0), angle(0,V), V > 3.14.

pick(s(X),spades):- \+ pick(X,hearts), spades(s(X)),
    angle(s(X),V), V > 3.14.
pick(s(X),clubs):- \+ pick(X,hearts), \+ spades(s(X)),
    clubs(s(X)), angle(s(X),V), V > 3.14.
pick(s(X),hearts):- \+ pick(X,hearts), \+ spades(s(X)),
    \+ clubs(s(X)), angle(s(X),V), V > 3.14.

at_least_once_spades :- pick(_,spades).
never_spades :- \+ at_least_once_spades.
```

With the query at_least_once_spades *we can compute the probability that the player picks at least one time spades when the axis of the wheel is between π (3.14) and 2π (6.28). The continuous random variables are represented by the second argument of predicate angle(T,X): they form a countable set, and there is a continuous random variable for each value of T (0, s(0), s(s(0)), ...). The set Y of discrete Boolean random variables is composed of $\{Y_i^c, Y_i^s \mid i = 1, 2, \dots\}$. Y_i^c (Y_i^s) represents clubs$(s^i(0))$ (spades$(s^i(0))$), and y_i^c (y_i^s) are values for Y_i^c (Y_i^s). Similarly, the set X of continuous random variables is composed of $\{X_i \mid i = 1, 2, \dots\}$. Each X_i has a range $[0, 2\pi]$, and its value is denoted with x_i. To compute the probability of this query, we can consider the mutually disjoint covering set of worlds $\omega = \omega_0 \cup \omega_1 \cup \dots$ where*

$$\omega_0 = \{(w_X, w_Y) \mid w_X = (x_1, x_2, \ldots), w_Y = (y_1^c, y_1^s, y_2^c, y_2^s, \ldots),$$
$$x_1 \in [\pi, 2\pi], y_1^s = 1\}$$
$$\omega_1 = \{(w_X, w_Y) \mid w_X = (x_1, x_2, \ldots), w_Y = (y_1^c, y_1^s, y_2^c, y_2^s, \ldots),$$
$$x_1 \in [\pi, 2\pi], y_1^s = 0, y_1^c = 1, x_2 \in [\pi, 2\pi], y_1^s = 1\}$$
$$\ldots$$

That is, for explanation ω_0 spades was selected at round 0 ($y_1^s = 1$) and the wheel (x_1) in the same round was in the range $[\pi, 2\pi]$. For explanation ω_1, spades was not selected at round 0 ($y_1^s = 0$), clubs was selected at round 0 ($y_1^c = 0$), the wheel (x_1) was in the range $[\pi, 2\pi]$ at round 0, spades was selected at round $s(0)$ ($y_2^s = 1$) and the wheel (x_2) was in the range $[\pi, 2\pi]$ at round $s(0)$. The probability for ω_0 can be computed as [3, 6]:

$$\mu(\omega_0) = \int_\pi^{2\pi} \mu_Y(\{(y_1^c, y_1^s, y_2^c, y_2^s, \ldots) \mid y_1^c = 1\}) \, d\mu_X$$
$$= \int_\pi^{2\pi} \frac{1}{3} \cdot \frac{1}{2\pi} dx_1 = \frac{1}{3} \cdot \frac{1}{2} = \frac{1}{6}.$$

where $\frac{1}{3}$ is the contribution of the discrete random variable (spades) and $\frac{1}{2\pi}$ is the contribution of the continuous one (angle). The probability for the other ω_i can be computed in a similar way. Overall, considering the limits, we get $\frac{1}{3} \cdot \frac{1}{2} \cdot \sum_{i=0}^{\infty}(\frac{2}{3} \cdot \frac{1}{2} \cdot \frac{1}{2})^i = \frac{1}{6} \cdot \sum_{i=0}^{\infty}(\frac{1}{6})^i = \frac{1}{6} \cdot \frac{6}{5} = \frac{1}{5}$ as probability for the query `at_least_once_spades`.

Example 4 (Gaussian mixture) *A Gaussian mixture model is a way to generate values of a continuous random variable: a discrete random variable is sampled and, depending on the sampled value, a different Gaussian distribution is selected for sampling the value of the continuous variable.*

A Gaussian mixture model with two components can be expressed in `cplint` *hybrid programs as* [1]*:*

```
h : 0.6.
heads :- h.
tails :- \+ h.
g(X) : gaussian(X, 0, 1).
h(X) : gaussian(X, 5, 2).
mix(X) :- heads, g(X).
mix(X) :- tails, h(X).
mix :- mix(X), X > 2.
```

The argument X of `mix(X)` *follows a distribution that is a mixture of two Gaussians, one with mean 0 and variance 1 with probability 0.6, and one with mean 5 and variance 2 with probability 1 - 0.6 = 0.4. We can then ask for the probability of* `mix`.

Here, predicates `g/1`, `h/1`, *and* `mix/1` *have a single argument which can hold continuous variables, so overall there is a finite set of continuous random variables. Since there are no term variables, each atom for these predicates in a world univocally determines its argument. For predicate* `mix/1` *this is not obvious as there are two clauses for it. However, the two clauses have mutually exclusive bodies, i.e., in each world only one of them is true. This property is further discussed in Section 4.*

---

[1] http://cplint.eu/e/gaussian_mixture.pl

Example 5 (Gaussian mixture and constraints, from [11]) *Consider a factory with two machines,* a *and* b*. Each machine produces a widget with a continuous feature. A widget is produced by machine* a *with probability 0.3 and by machine* b *with probability 0.7. If the widget is produced by machine* a*, the feature is distributed as a Gaussian with mean 2 and variance 1. If the widget is produced by machine* b*, the feature is distributed as a Gaussian with mean 3 and variance 1. The widget then is processed by a third machine that adds a random quantity to the feature. The quantity is distributed as a Gaussian with mean 0.5 and variance 1.5. This can be encoded by in* cplint *hybrid programs as* [2]*:*

```
machine(a) : 0.3.
machine(b) :- \+ machine(a).
st(a,Z) : gaussian(Z, 2, 1).
st(b,Z) : gaussian(Z, 3, 1).
pt(Y) : gaussian(Y, 0.5, 1.5).
widget(X) :- machine(M), st(M,Z), pt(Y), X =:= Y + Z.
ok_widget :- widget(X), X > 1.0.
```

We can then ask the probability of ok_widget*.*

Here, X*,* Y*, and* Z *are continuous variables and* M *is a term variable. Since* X *is a continuous variable, in every world there should be a single value for* X *that makes* widget(X) *true. Predicate* widget/1 *has a single clause but the clause has two groundings, one for* M = a*, and one for* M = b*, so in principle there could be two values for* X *in true groundings of* widget(X)*. However, as in Example 4, the two groundings of the rule have mutually exclusive bodies, as in each world either* machine(a) *is true or* machine(b) *is true, but not both.*

Example 6 (Estimation of the mean of a Gaussian) *Consider now this example* [3]*:*

```
mean(M) : gaussian(M,1,5).
value(_,M,X) : gaussian(X,M,2).
value(I,X) :- mean(M), value(I,M,X).
```

The program states that, for an index I*, the continuous variable* X *is sampled from a Gaussian whose variance is 2 and whose mean* M *is sampled from a Gaussian with mean 1 and variance 5.*

This program can be used to estimate the mean of a Gaussian by querying mean(M) *given observations for atom* value(I,X) *for different values of* I*.*

Here, the first argument of value/3 *can hold a term variable while its second and third arguments can hold a continuous variable. The second argument is used as a parameter in the probability density of the third argument. It is not immediate to see whether this program has a well-defined semantics or not. In fact, the semantics does not allow specifying the parameters of continuous distributions with values computed by the program, but we can consider continuous variables* M *and* X *as specified by a joint density (see Section 4). Since a Gaussian density with a Gaussian mean is still a Gaussian, the joint density will be a multivariate Gaussian.*

In the next Section, we illustrate more in detail the syntactic requirements that hybrid probabilistic logic programs, and thus probabilistic constraint logic programs, must follow to have a well-defined semantics.

---

[2]http://cplint.eu/e/widget.pl

[3]http://cplint.eu/example/inference/gauss_mean_est.pl

4 Syntactic Requirements

To preserve the well-definedness, we require that the set of random variables must be countable. That is, every random variable must be associated with a ground logical atom. The discrete arguments of this logical atom can contain terms and cannot be real values. Let us now analyse several possible situations.

In the fourth clause of Example 3, (reported here for clarity)

```
pick(0,spades) :- spades(0), angle(0,V), V > 3.14.
```

the first argument of the continuous random variable `angle` is a constant. Similarly happens with the clauses with `angle(s(X),V)`. Variable `X` appears in the head, so it is ground when `angle(s(X),V)` is called (during the recursion or by directly set a value for `X` in the query), and the continuous random variable `V` is associated with term `s(X)` by predicate `angle/2`. In both cases, `s(X)` is a ground logical term (integer in case of 0), so the semantics is well-defined.

Consider now Example 4: in this program there are two random variables, identified respectively by `g(X)` and `h(X)`. Predicate `mix/1` is composed of two clauses with the same head but different bodies, a situation that may lead to an ill-defined program where both clauses are true and the random variable `X` is defined by two different distributions. However, the two bodies are mutually exclusive: the discrete random variable `h` is used to discriminate between the two. When `h` is true, `heads` is true, and thus the first clause of `mix/1` is considered. When it is false, the second `mix/1` clause is considered. This mutual exclusivity guarantees the well-definedness of the program. Algorithmically, this property can be verified performing clause unfolding, obtaining two clauses with the same head with bodies mutually exclusive, since they contain at least one atom in common, but in one it is positive, in the other it is negated. The same happens for Example 5. This idea can be extended to the case where there are *n* clauses with the same head defining *n* different distributions for the same variable, provided that all the clauses are mutually exclusive.

To see a counterexample, if we modify the clauses for predicate `mix/1` of Example 4 in this way:

```
mix(X) :- g(X).
mix(X) :- h(X).
```

the program is ill-defined, since the two clauses are not mutually exclusive, and there is not a single distribution for variable `X`.

Focus now on Example 6. Here, atom `value/3` has a continuous random variable (`M`) as input. To keep the well-definedness, this input must be defined in another continuous probabilistic fact and must be used only as a parameter for another distribution. In fact, `M` is obtained from a Gaussian distribution and it is used as a parameter for the mean for another Gaussian distribution. In this way, variables `M` and `X` are specified by a joint density. Note that the first variable of `value/3` is a term variable and can be associated with a ground atom (integer, for example), preserving the well-definedness: the value of the term variable uniquely identifies the value of the continuous random variable. In this way, the position in the sequence of random variables is exactly identified.

This situation can be straightforwardly extended to programs in which there are multiple continuous random variables as input: for instance, in Example 6, the variance for `X` in clause `value/3` can be sampled from another distribution instead of being fixed to 2.

In other cases, when the value of a continuous random variable is used as variable for another term, but not as a parameter for a distribution, the semantics is ill-defined. Consider the following modification to Example 6:

```
value(X) : gaussian(X,1,5).
value(_,M) : gaussian(M,2,2).
res(M) :- value(X), value(X,M).
```

Here, X in clause `res/1` is the value of a continuous random variable but it is not used as parameter for the distribution of M. The continuous random variable M cannot be associated with a term that uniquely identifies it, so the program is ill-defined. The difference with Example 6 is that in Example 6 the term variable used as index (I) can be associated with a ground logical term, while here it is associated with a real value (X). If in this last example the variable X is a discrete random variable rather than a continuous one, the program would be well-defined, since X is associated with a ground logical atom.

To see how to compute the probability in this situation, consider the following simple program (a simplification of Example 6, where Gaussian distribution are replaced by Uniform distributions identified by `uniform_dens/3`):

Example 7 `angle_a(_,X,Y) : uniform_dens(Y,X,2).`
```
angle_b(X) : uniform_dens(X,0,1).
success(I) :- angle_b(X), angle_a(I,X,Y), Y < 1.5.
```

Here, the variable X in `angle_b/1` follows a uniform distribution between 0 and 1. For variable Y in `angle_a/1`, its lower bound is sampled from another uniform distribution. The probability distribution defined by the two random variables can be considered as a joint distribution between the two, so they can be treated as one multivariate random variable in the sequence of random variables, indexed by the logical term unified with I. Their joint density function is given by

$$f_{XY}(x,y) = \frac{1}{1-0} \cdot \frac{1}{2-x} = \frac{1}{2-x}$$

and thus the probability of the query `success(0)` is

$$P(success(0)) = P[Y < 1.5] = \int_0^1 \frac{1}{2-x} \left(\int_x^{1.5} 1\, dy \right) dx =$$
$$= \int_0^1 \frac{1.5-x}{2-x}\, dx = 1 - \frac{ln(2)}{2} \approx 0.653$$

As said in Section 2, the range-restrictedness property ensures that the answers of a query are always ground instantiations of it. However, there can be some exceptions. Consider the first three lines of Example 3, reported here for clarity:

```
1/3 :: spades(_).
1/2 :: clubs(_).
angle(_,X) : uniform_dens(X,0,6.28).
```

Both discrete and continuous probabilistic facts are not range restricted, since they contain an input variable (anonymous) that is not ground. To ensure that the answers of a query are always ground instantiations of it, we need to make sure that, when these types of facts are called, all the input variables are ground. We then impose that the arguments of probabilistic facts that are not ground must be variables that appear in previous literal in the body: the preceding calls will bind these variables, so the probabilistic facts will be called with all input arguments ground, and a probability value will be assigned to a ground query. The order of the terms is fundamental: the variables in a probabilistic fact must appear in a preceding literal, to ensure the well-definedness. Consider the following clarifying example:

```
a(1).
f(_,X) : uniform_dens(X,0,6).


g0(X):- a(X), f(X,V), V > 1.
g1(X):- f(X,V), V > 1, a(X).
```

For the query `g0(X)`, when `f(X,Y)` is called, `X` has already been substituted with 1, and `f/2` has a ground input variable. This does not hold for query `g1(X)` (even if the terms in the body are the same, they are in different order), since `f/2` is called with the input variable not instantiated, thus violating the syntactic requirements.

    To sum up, if we consider the following simple program:

```
ud(_,V) : uniform_dens(V,0,6).
1/2 :: a(_).
b(X,1):- a(X).
b(X,2):- \+a(X).
n(1).
f_0(X):- a(X).
f_1:- a(1).
f_2(X):- n(X), a(X).
f_3:- a(_).
f_4(X):- b(X,V), a(V),
f_5:- ud(1,V), V > 2.
```

the answers of the queries `f_0(1)`, `f_1`, `f_2(X)`, and `f_5` are ground instantiations of them, since the input term variable for probabilistic facts `a/1` and `ud/2` is bounded to 1 in all the queries. Similarly happens for `f_4(1)`, where `V` can unify with 1 or 2, depending on the truth value of `a`. On the contrary, the answers for `f_0(_)`, `f_3`, and `f_4(_)` are not ground instantiations, since the term variable of the probabilistic fact `a` is not ground (for the first two queries) or the input variable `X` of `b/2` is not ground (for the third query).

5   Conclusions

In this paper, following the semantics proposed in [3], we delineated a precise syntax and explained the necessary syntactic conditions to maintain the well-definedness for hybrid probabilistic logic programs. We covered several cases in which ill-definedness may arise, providing examples and counterexamples. As future work, we plan to develop algorithms to automatically check these properties as well as developing new inference algorithms that can manage infinite domains [5].

References

[1] Marco Alberti, Elena Bellodi, Giuseppe Cota, Fabrizio Riguzzi & Riccardo Zese (2017): *cplint on SWISH: Probabilistic Logical Inference with a Web Browser*. Intelligenza Artificiale 11(1), pp. 47–64, doi:10.3233/IA-170105.

[2] Damiano Azzolini, Fabrizio Riguzzi & Evelina Lamma (2019): *Studying Transaction Fees in the Bitcoin Blockchain with Probabilistic Logic Programming*. Information 10(11), p. 335, doi:10.3390/info10110335.

[3] Damiano Azzolini, Fabrizio Riguzzi & Evelina Lamma (2021): *A Semantics for Hybrid Probabilistic Logic Programs with Function Symbols*. Artificial Intelligence 294, p. 103452, doi:10.1016/j.artint.2021.103452.

[4] Damiano Azzolini, Fabrizio Riguzzi, Evelina Lamma & Franco Masotti (2019): *A Comparison of MCMC Sampling for Probabilistic Logic Programming*. In Mario Alviano, Gianluigi Greco & Francesco Scarcello, editors: *Proceedings of the 18th Conference of the Italian Association for Artificial Intelligence (AI*IA2019), Rende, Italy 19-22 November 2019*, Lecture Notes in Computer Science, Springer, Heidelberg, Germany, doi:10.1007/978-3-030-35166-3_2.

[5] Vaishak Belle (2020): *Symbolic Logic Meets Machine Learning: A Brief Survey in Infinite Domains*. In Jesse Davis & Karim Tabia, editors: *Scalable Uncertainty Management*, Springer International Publishing, Cham, pp. 3–16, doi:10.1007/978-3-030-58449-8_1.

[6] Y.S. Chow & H. Teicher (2012): *Probability Theory: Independence, Interchangeability, Martingales*. Springer Texts in Statistics, Springer.

[7] Fabio Gagliardi Cozman & Denis Deratani Mauá (2017): *On the Semantics and Complexity of Probabilistic Logic Programs*. Journal of Artificial Intelligence Research 60, pp. 221–262, doi:10.1613/jair.5482.

[8] Luc De Raedt & Angelika Kimmig (2015): *Probabilistic (Logic) Programming Concepts*. Machine Learning 100(1), pp. 5–47, doi:10.1007/s10994-015-5494-z.

[9] Luc De Raedt, Angelika Kimmig & Hannu Toivonen (2007): *ProbLog: A Probabilistic Prolog and Its Application in Link Discovery*. In Manuela M. Veloso, editor: *20th International Joint Conference on Artificial Intelligence (IJCAI 2007)*, 7, AAAI Press/IJCAI, pp. 2462–2467. Available at `http://www.ijcai.org/papers07/Papers/IJCAI07-396.pdf`.

[10] Bernd Gutmann, Manfred Jaeger & Luc De Raedt (2011): *Extending ProbLog with Continuous Distributions*. In Paolo Frasconi & Francesca A. Lisi, editors: *20th International Conference on Inductive Logic Programming (ILP 2010)*, LNCS 6489, Springer, pp. 76–91, doi:10.1007/978-3-642-21295-6_12.

[11] Muhammad Asiful Islam, CR Ramakrishnan & IV Ramakrishnan (2012): *Inference in probabilistic logic programs with continuous random variables*. Theory and Practice of Logic Programming 12, pp. 505–523, doi:10.1017/S1471068412000154.

[12] Steffen Michels, Arjen Hommersom, Peter J. F. Lucas & Marina Velikova (2015): *A new probabilistic constraint logic programming language based on a generalised distribution semantics*. Artificial Intelligence 228, pp. 1–44, doi:10.1016/j.artint.2015.06.008.

[13] Stephen Muggleton (2000): *Learning Stochastic Logic Programs*. In Lise Getoor & David Jensen, editors: *Learning Statistical Models from Relational Data, Papers from the 2000 AAAI Workshop, AAAI Workshops* WS-00-06, AAAI Press, pp. 36–41.

[14] Arnaud Nguembang Fadja & Fabrizio Riguzzi (2017): *Probabilistic Logic Programming in Action*. In Andreas Holzinger, Randy Goebel, Massimo Ferri & Vasile Palade, editors: *Towards Integrative Machine Learning and Knowledge Extraction*, LNCS 10344, Springer, doi:10.1007/978-3-319-69775-8_5.

[15] David Poole (1997): *The Independent Choice Logic for Modelling Multiple Agents Under Uncertainty*. Artificial Intelligence 94(1-2), pp. 7–56, doi:10.1016/S0004-3702(97)00027-1.

[16] Fabrizio Riguzzi (2013): *MCINTYRE: A Monte Carlo System for Probabilistic Logic Programming*. Fundamenta Informaticae 124(4), pp. 521–541, doi:10.3233/FI-2013-847.

[17] Fabrizio Riguzzi (2016): *The Distribution Semantics for Normal Programs with Function Symbols*. International Journal of Approximate Reasoning 77, pp. 1–19, doi:10.1016/j.ijar.2016.05.005.

[18] Fabrizio Riguzzi (2018): *Foundations of Probabilistic Logic Programming*. River Publishers, Gistrup, Denmark.

[19] Fabrizio Riguzzi, Elena Bellodi, Evelina Lamma, Riccardo Zese & Giuseppe Cota (2016): *Probabilistic Logic Programming on the Web*. Software: Practice and Experience 46(10), pp. 1381–1396, doi:10.1002/spe.2386.

[20] Fabrizio Riguzzi & Terrance Swift (2013): *Well-Definedness and Efficient Inference for Probabilistic Logic Programming under the Distribution Semantics*. Theory and Practice of Logic Programming 13(2), pp. 279–302, doi:10.1017/S1471068411000664.

[21] Taisuke Sato (1995): *A Statistical Learning Method for Logic Programs with Distribution Semantics*. In Leon Sterling, editor: *Logic Programming, Proceedings of the Twelfth International Conference on Logic Programming, Tokyo, Japan, June 13-16, 1995*, MIT Press, pp. 715–729.

[22] A. Van Gelder, K. A. Ross & J. S. Schlipf (1991): *The Well-founded Semantics for General Logic Programs*. Journal of the ACM 38(3), pp. 620–650, doi:10.1145/116825.116838.

[23] J. Vennekens, Marc Denecker & Maurice Bruynooghe (2009): *CP-logic: A language of causal probabilistic events and its relation to logic programming*. Theory and Practice of Logic Programming 9(3), pp. 245–308, doi:10.1017/S1471068409003767.

[24] Joost Vennekens, Sofie Verbaeten & Maurice Bruynooghe (2004): *Logic Programs With Annotated Disjunctions*. In Bart Demoen & Vladimir Lifschitz, editors: *20th International Conference on Logic Programming (ICLP 2004)*, *LNCS* 3131, Springer, pp. 431–445, doi:10.1007/978-3-540-27775-0_30.

[25] Pedro Zuidberg Dos Martires, Anton Dries & Luc De Raedt (2018): *Knowledge Compilation with Continuous Random Variables and its Application in Hybrid Probabilistic Logic Programming*. CoRR abs/1807.00614.