Explanations as Programs in Probabilistic Logic Programming[⋆]

Germán Vidal

VRAIN, Universitat Politècnica de València, Spain
gvidal@dsic.upv.es

Abstract. The generation of comprehensible explanations is an essential feature of modern artificial intelligence systems. In this work, we consider *probabilistic logic programming*, an extension of logic programming which can be useful to model domains with relational structure and uncertainty. Essentially, a program specifies a probability distribution over possible *worlds* (i.e., sets of facts). The notion of *explanation* is typically associated with that of a world, so that one often looks for the *most probable world* as well as for the worlds where the query is true. Unfortunately, such explanations exhibit no causal structure. In particular, the chain of inferences required for a specific prediction (represented by a query) is not shown. In this paper, we propose a novel approach where explanations are represented as programs that are generated from a given query by a number of unfolding-like transformations. Here, the chain of inferences that proves a given query is made explicit. Furthermore, the generated explanations are minimal (i.e., contain no irrelevant information) and can be parameterized w.r.t. a specification of *visible* predicates, so that the user may hide uninteresting details from explanations.

Published as *Vidal, G. (2022). Explanations as Programs in Probabilistic Logic Programming. In: Hanus, M., Igarashi, A. (eds) Functional and Logic Programming. FLOPS 2022. Lecture Notes in Computer Science, vol 13215. Springer, Cham.*

The final authenticated publication is available online at
`https://doi.org/10.1007/978-3-030-99461-7_12`

1 Introduction

Artificial inteligence (AI) and, especially, machine learning systems are becoming ubiquitous in many areas, like medical diagnosis [9], intelligent transportation [33], or different types of recommendation systems [26], to name a few. While

⋆ This work has been partially supported by the EU (FEDER) and the Spanish MCI under grant PID2019-104735RB-C41/ AEI/10.13039/501100011033 (SAFER), by the *Generalitat Valenciana* under grant Prometeo/2019/098 (DeepTrust), and by TAILOR, a project funded by EU Horizon 2020 research and innovation programme under GA No 952215.

prediction errors are sometimes acceptable, there are areas where blindly following the assessment of an AI system is not desirable (e.g., medical diagnosis). In these cases, generating *explanations* that are comprehensible by non-expert users would allow them to verify the reliability of the prediction as well as to improve the system when the prediction is not correct. Furthermore, the last regulation on data protection in the European Union [10] has introduced a "right to explanation" for algorithmic decisions. All in all, the generation of comprehensible explanations is an essential feature of modern AI systems.

Currently, there exist many approaches to *explainable AI* (XAI) [3], which greatly differ depending on the considered application. In particular, so-called *interpretable machine learning* [18] puts the emphasis on the interpretability of the models and their predictions. In this work, we consider *probabilistic logic programming* (PLP) [24], which can be useful to model domains with relational structure and uncertainty. PLP has been used for both inference—e.g., computing the marginal probability of a set of random variables given some evidence—and learning [30,11]. Among the different approaches to PLP, we consider those that are based on Sato's *distribution semantics* [29]. This is the case of several proposals that combine logic programming and probability, like Logic Programs with Annotated Disjunctions (LPADs) [32], ProbLog [25], Probabilistic Horn Abduction (PHA) [22], Independent Choice Logic (ICL) [23], and PRISM [30].

In particular, we consider the ProbLog approach for its simplicity, but we note that the expressive power of all languages mentioned above is the same (see, e.g., [27, Chapter 2]). A ProbLog program extends a logic program with a set of *probabilistic facts*. A probabilistic fact has the form $p :: a$ and denotes a random variable which is true with probability p and false with probability $1 - p$. Here, a program defines a probability distribution over *worlds*, i.e., sets of (possibly negated) atoms corresponding to the probabilistic facts of the program. Essentially, the probability of a world is equal to the product of the probabilities of its true and false facts, while the probability of a query is computed by *marginalization*, i.e., by summing up the probabilities of the worlds where the query is true.

The notion of *explanation* of a query is often associated with that of a world. For instance, the MPE task [31], which stands for *Most Probable Explanation*, consists in finding the world with the highest probability. However, a world exhibits no causal structure and, thus, it is up to the user to understand *why* the given collection of facts actually allow one to infer a particular prediction (it might even be counterintuitive; see Example 4). Moreover, a world typically contains facts whose truth value is irrelevant for the query, which might be an additional source of confusion. Alternatively, one could consider a *proof tree* of a query as an explanation. While the chain of inferences and the links to the query are now explicit, proof trees are typically very large and can be complex to understand by non-experts.

In this paper, we propose a novel approach where explanations are represented as programs that are generated from a given query by a number of unfolding-like transformations. In this way, we have the same advantages of using proof trees as explanations (the chain of inferences and the link to the query are explicit), but they are often easier to understand by non-experts because of the following reasons: first, an explanation is associated with a *single* proof, so it is conceptually simpler than a proof tree (that might comprise several proofs); second, facts and rules have a more intuitive reading than a proof tree (and could easily be represented using natural language); finally, the generated explanations can be parameterized w.r.t. a set of *visible* predicates. If no predicate is visible, our explanations are not very different from a (partial) world, since they just contain the probabilistic facts that make a query true in a particular proof. On the other hand, if all predicates are visible, the computation of an explanation essentially boils down to computing the (relevant) grounding of a program for a given proof of the query. It is thus up to the user to determine the appropriate level of detail in the explanation so that only the most interesting relations are shown.

In this work, we present a constructive algorithm for generating the explanations of a query such that the following essential properties hold: i) the probability of each proof is preserved in the corresponding explanation, ii) explanations do not contain facts that are irrelevant for the given query, and iii) the (marginal) probability of the query in the original program is equivalent to that in the union of the computed explanations. In order to check the viability of the approach, we have developed a proof-of-concept implementation, `xgen`,[1] that takes a ProbLog program and a query and produces a set of explanations for this query, together with their associated probabilities.

2 Probabilistic Logic Programming (PLP)

In this section, we briefly introduce PLP following the ProbLog approach (see, e.g., [11,14,24,25,27] for a detailed account).

Let us first recall some basic terminology from logic programming [1,15]. We consider a first-order language with a fixed vocabulary of predicate symbols, constants, and variables denoted by Π, $\mathcal{C}$ and $\mathcal{V}$, respectively.[2] An *atom* has the form $f(t_1, \ldots, t_n)$ with $f/n \in \Pi$ and $t_i \in (\mathcal{C} \cup \mathcal{V})$ for $i = 1, \ldots, n$. A *definite clause* has the form $h \leftarrow B$, where h (the *head*) is an atom and B (the *body*) is a conjunction of atoms, typically denoted by a sequence $a_1, \ldots, a_n$, $n \geq 0$; if $n = 0$, the clause is called a *fact* and denoted by h; otherwise ($n > 0$), it is called a *rule*. A *query* is a clause with no head, and is denoted by a sequence of atoms. $\mathsf{var}(s)$ denotes the set of variables in the syntactic object s, where s can be an atom, a query or a clause. A syntactic object s is *ground* if $\mathsf{var}(s) = \emptyset$. Substitutions and their operations are defined as usual; the application of a substitution θ to a syntactic object s is denoted by juxtaposition, i.e., we write $s\theta$ rather than $\theta(s)$.

In the following, we consider that $\Pi = \Pi_p \uplus \Pi_d$, where Π_p are the *probabilistic predicates* and Π_d are the *derived predicates*, which are disjoint. An atom $f(t_1, \ldots, t_n)$ is called a *probabilistic atom* if $f \in \Pi_p$ and a *derived atom* if

[1] Publicly available from `https://github.com/mistupv/xgen` .

[2] We do not consider function symbols in this work.

$f \in \Pi_d$. A *probabilistic logic program* (or just *program* when no confusion can arise) $\mathcal{P} = \mathcal{P}_p \uplus \mathcal{P}_d$ consists of a set of ground probabilistic facts $\mathcal{P}_p$ and a set of definite clauses $\mathcal{P}_d$. A *probabilistic fact* has the form $p :: a$, where a is a ground atom and $p \in [0,1]$ is a probability such that a is true with probability p and false with probability $1-p$. These ground facts represent the Boolean random variables of the model, which we assume mutually independent.

In this paper, we also allow *nonground* probabilistic facts, that are replaced by their finite groundings using the Herbrand universe, i.e., using the constants from $\mathcal{C}$. More generally, we consider *intensional* probabilistic facts defined by (probabilistic) clauses of the form $p :: f(x_1,\ldots,x_n) \leftarrow B$, where B only contains derived atoms. Such a rule represents the set of ground probabilistic facts $p :: f(x_1,\ldots,x_n)\theta$ such that $B\theta$ is true in the underlying model.

Example 1. Consider the following program (a variation of an example in [11]):[3]

```
0.8::stress(X) :- person(X).        person(ann).
0.3::influences(bob,carl).          person(bob).
smokes(X) :- stress(X).
smokes(X) :- influences(Y,X),smokes(Y).
```

Here, we have two probabilistic predicates, `stress/1` and `influences/2`, and a logic program that defines the relation `smokes/1`. Basically, the program states that a person (either `ann` or `bob`) is stressed with probability `0.8`, `bob` influences `carl` with probability `0.3`, and that a person smokes either if (s)he is stressed or is influenced by someone who smokes.

Observe that the first probabilistic clause is equivalent to the following set of ground probabilistic facts: `0.8::stress(ann)`, `0.8::stress(bob)`.

We note that probabilistic clauses can always be rewritten to a combination of probabilistic facts and non-probabilistic clauses [11]. For instance, the probabilistic clause in the example above could be replaced by

```
0.8::p(X).       stress(X) :- person(X),p(X).
```

In this work, we assume that the Herbrand universe is finite (and coincides with the domain $\mathcal{C}$ of constants) and, thus, the set of ground instances of each probabilistic fact is finite.[4] Given a program $\mathcal{P}$, we let $G(\mathcal{P})$ denote the set of its *ground* probabilistic facts (after grounding nonground probabilistic and intensional facts, if any). An *atomic choice* determines whether a ground probabilistic fact is chosen or not. A *total choice* makes a selection for *all* ground probabilistic facts; it is typically represented as a set of ground probabilistic atoms (the ones that are true). Note that, given n ground probabilistic atoms, we have 2^n possible total choices.

A program $\mathcal{P}$ then defines a probability distribution over the total choices. Moreover, since the random variables associated with the ground probabilistic

[3] We follow Prolog's notation in examples: variables start with an uppercase letter and the implication "$\leftarrow$" is denoted by "`:-`".

[4] See Sato's seminal paper [29] for the distribution semantics in the infinite case.

		$P(w_i)$
w_1	{`stress(ann)`,`stress(bob)`,`influences(bob, carl)`}	$0.8 \cdot 0.8 \cdot 0.3 = 0.192$
w_2	{`stress(ann)`,`stress(bob)`}	$0.8 \cdot 0.8 \cdot 0.7 = 0.448$
w_3	{`stress(ann)`, `influences(bob, carl)`}	$0.8 \cdot 0.2 \cdot 0.3 = 0.048$
w_4	{`stress(ann)`}	$0.8 \cdot 0.2 \cdot 0.7 = 0.112$
w_5	{`stress(bob)`,`influences(bob, carl)`}	$0.2 \cdot 0.8 \cdot 0.3 = 0.048$
w_6	{`stress(bob)`}	$0.2 \cdot 0.8 \cdot 0.7 = 0.112$
w_7	{`influences(bob, carl)`}	$0.2 \cdot 0.2 \cdot 0.3 = 0.012$
w_8	{ }	$0.2 \cdot 0.2 \cdot 0.7 = 0.028$

Fig. 1. Possible worlds for Example 1

facts are mutually independent, the probability of a total choice $L \subseteq G(\mathcal{P})$ can be obtained from the product of the probabilities of its atomic choices:

$$P(L) = \prod_{a\theta \in L} \pi(a) \quad \cdot \prod_{a\theta \in G(\mathcal{P}) \setminus L} 1 - \pi(a)$$

where $\pi(a)$ denotes the probability of the fact, i.e., $\pi(a) = p$ if $p :: a \in \mathcal{P}_p$. A possible *world* is then defined as the least Herbrand model of $L \cup \mathcal{P}_d$, which is unique. Typically, we denote a world by a total choice, omitting the (uniquely determined) truth values for derived atoms. By definition, the sum of the probabilities of all possible worlds is equal to 1.

In the following, we only consider *atomic* queries. Nevertheless, note that an arbitrary query B could be encoded using an additional clause of the form $q \leftarrow B$. The probability of a query q in a program $\mathcal{P}$, called the *success probability* of q in $\mathcal{P}$, in symbols $P(q)$, is defined as the marginal of $P(L)$ w.r.t. query q:

$$P(q) = \sum_{L \subseteq G(\mathcal{P})} P(q|L) \cdot P(L)$$

where $P(q|L) = 1$ if there exists a substitution θ such that $L \cup \mathcal{P}_d \models q\theta$ and $P(q|L) = 0$ otherwise. Intuitively speaking, the success probability of a query is the sum of the probabilities of all the worlds where this query is provable.[5]

Example 2. Consider again the program in Example 1. Here, we have eight possible worlds, which are shown in Figure 1. Observe that the sum of the probabilities of all worlds is 1. Here, the query `smokes(carl)` is true in worlds w_1 and w_5. Thus, its probability is $0.192 + 0.048 = 0.24$.

Since the number of worlds is finite, one could compute the success probability of a query by enumerating all worlds and, then, checking whether the query is true in each of them. Unfortunately, this approach is generally unfeasible in practice due to the large number of possible worlds. Instead, a combination of inference and a conversion to a Boolean formula is often used (see, e.g., [11]).

[5] Equivalently, has a successful SLD derivation; see Section 3.1 for a precise definition of SLD (*Selective Linear Definite clause*) resolution.

3 Explanations as Programs

In this section, we focus on the notion of explanation in the contect of PLP. Here, we advocate that a *good* explanation should have the following properties:

- *Causal structure.* An explanation should include the chain of inferences that supports a given prediction. It is not sufficient to just show a collection of facts. It should answer *why* a given query is true, so that the user can follow the reasoning from the query back to the considered probabilistic facts.
- *Minimality.* An explanation should not include irrelevant information. In particular, those facts whose truth value is indifferent for a given query should not be part of the explanation.
- *Understandable.* The explanation should be easy to follow by non-experts in PLP. Moreover, it is also desirable for explanations to be parametric w.r.t. the information that is considered relevant by the user.

In the following, we briefly review some possible notions of an explanation and, then, introduce our new proposal.

3.1 Explanations in PLP

Typically, *explanations* have been associated with *worlds*. For instance, the MPE (*Most Probable Explanation*) task [31] consists in finding the world with the highest probability given some *evidence* (in our context, given that some query is true). However, a world does not show the chain of inferences of a given query and, moreover, it is not minimal by definition, since it usually includes a (possibly large) number of probabilistic facts whose truth value is irrelevant for the query.

Alternatively, one can consider a probabilistic logic program itself as an explanation. Here, the causal structure is explicit (given by the program clauses). Moreover, derived rules are easy to understand or can easily be explained using natural language. However, the program explains the complete *model* but it is not so useful to explain a particular query: the chain of inferences is not obvious and, moreover, programs are not usually minimal since they often contain a (possibly large) number of facts and rules which are not relevant for a particular query.

Another alternative consists in using the *proof of a query* as an explanation. Following [14], one can associate a *proof* of a query with a (minimal) *partial* world w' such that for all worlds $w \supseteq w'$, the query is true in w. In this case, one can easily ensure minimality (e.g., by using SLD resolution to determine the ground probabilistic atoms that are needed to prove the query). However, even if the partial world contains no irrelevant facts, it is still not useful to determine the chain of inferences behind a given query. In order to avoid this shortcoming, one could represent the proofs of a query by means of an SLD tree. Let us further explore this possibility.

First, we recall some background from logic programming [15]. Given a logic program $\mathcal{P}$, we say that $B_1, a, B_2 \leadsto_\theta (B_1, B, B_2)\theta$ is an *SLD resolution step* if

$h \leftarrow B$ is a renamed apart clause (i.e., with fresh variables) of program $\mathcal{P}$, in symbols, $h \leftarrow B \ll \mathcal{P}$, and $\theta = \mathsf{mgu}(a, h)$ is the *most general unifier* of atoms a and h. An *SLD derivation* is a (finite or infinite) sequence of SLD resolution steps. As is common, $\leadsto^*$ denotes the reflexive and transitive closure of $\leadsto$. In particular, we denote by $A_0 \leadsto^*_\theta A_n$ a derivation $A_0 \leadsto_{\theta_1} A_1 \leadsto_{\theta_2} \ldots \leadsto_{\theta_n} A_n$, where $\theta = \theta_1 \ldots \theta_n$ if $n > 0$ (and $\theta = id$ otherwise). An SLD derivation is called *successful* if it ends with the query *true* (an empty conjunction), and it is called *failed* if there is an atom that does not unify with the head of any clause. Given a successful SLD derivation $A \leadsto^*_\theta \mathit{true}$, the associated *computed answer*, $\theta\!\upharpoonright_{\mathsf{var}(A)}$, is the restriction of θ to the variables of the initial query A. SLD derivations are represented by a (possibly infinite) finitely branching tree called *SLD tree*.

All the previous notions (SLD step, derivation and tree, successful derivation, computed answer, etc) can be naturally extended to deal with probabilistic logic programs by simply ignoring the probabilities in probabilistic clauses.

Following [14], the probability of a single proof is the marginal over all programs where such a proof holds. Thus, it can be obtained from the product of the probabilities of the ground probabilistic facts used in the corresponding SLD derivation.[6] In principle, one could first apply a grounding stage—where all non-ground and intensional probabilistic facts are replaced by ground probabilistic facts—and, then, apply the above definition. Often, only a partial grounding is required (see, e.g., [11]). Since grounding is orthogonal to the topics of this paper, in the following we assume that the following property holds: for each considered successful SLD derivation $q \leadsto^*_\theta \mathit{true}$ that uses probabilistic clauses (i.e., probabilistic facts and rules) $p_1 :: c_1, \ldots, p_n :: c_n$, we have that $c_1\theta, \ldots, c_n\theta$ are ground, i.e., it suffices if the probabilistic clauses used in the derivation *become eventually* ground.

In practice, *range-restrictedness* is often required for ensuring that all probabilistic facts become eventually ground in an SLD derivation, where a program is range-restricted if all variables in the head of a clause also appear in some atom of the body [28]. Moreover, one can still allow some probabilistic facts with non-ground arguments (which are not range-restricted) as long as they are called with a ground term in these arguments; see [4, Theorem 1]. A similar condition is required in ProbLog, where a program containing a probabilistic fact of the form `0.6::p(X)` is only acceptable if the query bounds variable `X`, e.g., `p(a)`. However, if the query is also non-ground, e.g., `p(X)`, then ProbLog outputs an error: "Encountered a non-ground probabilistic clause".[7]

In the following, given a successful SLD derivation $D = (q \leadsto^*_\theta \mathit{true})$, we let *prob_facts*$(D)$ be the set of ground probabilistic clauses used in D, i.e., $c\theta$ for each probabilistic clause c used in D. The probability of a proof (represented by a successful SLD derivation) can then be formalized as follows:

[6] Observe that each fact should only be considered once. E.g., given a successful SLD derivation that uses the ground probabilistic fact $\mathtt{0.4} :: \mathtt{person(ann)}$ twice, the associated probability is $\mathtt{0.4}$ rather than $\mathtt{0.4} \cdot \mathtt{0.4} = \mathtt{0.16}$.

[7] The interested reader can try the online ProbLog interpreter at `https://dtai.cs.kuleuven.be/problog/editor.html`.

Definition 1 (probability of a proof). *Let $\mathcal{P}$ be a program and D a successful SLD derivation for some (atomic) query q in $\mathcal{P}$. The probability of the proof represented by D is obtained as follows: $P(D) = \Pi_{c\theta \in \mathit{prob\_facts}(D)}\ \pi(c)$.*

Let us illustrate this definition with an example:

Example 3. Consider again our running example. Here, we have the following successful SLD derivation D for the query `smokes(carl)`:[8]

$$\begin{array}{lll}
\mathtt{smokes(carl)} & \leadsto & \mathtt{influences(Y,carl),smokes(Y)} \\
 & \leadsto_{\{\mathtt{Y/bob}\}} & \mathtt{smokes(bob)} \\
 & \leadsto & \mathtt{stress(bob)} \\
 & \leadsto & \mathit{true}
\end{array}$$

Here, $\mathit{prob\_facts}(D) = \{\mathtt{influence(bob,carl)},\ \mathtt{stress(bob)\ {:}\text{-}\ person(bob)}\}$, whose probabilities are `0.3` and `0.8`, respectively. Hence, $P(D) = \mathtt{0.3} \cdot \mathtt{0.8} = \mathtt{0.24}$.

One might think that the probability of a query can then be computed as the sum of the probabilities of its successful derivations. This is not generally true though, since the successful derivations may *overlap* (e.g., two successful derivations may use some common probabilistic facts). Nevertheless, several techniques use the SLD tree as a first stage to compute the success probability (see, e.g., [14,11]).

Computing the most likely proof of a query attracted considerable interest in the PLP field (where it is also called *Viterbi proof* [14]). Here, one aims at finding the most probable partial world that entails the query (which can be obtained from the proof with the highest probability). Note that, although it may seem counterintuitive, the MPE cannot always be obtained by extending the most likely proof of a query, as the following example illustrates:

Example 4. Consider the following program from [31, Example 6]:

```
0.4::red.    0.9::green.     win :- red, green.
0.5::blue.   0.6::yellow.    win :- blue, yellow.
```

Here, `win` has two proofs: one uses the probabilistic facts `red` and `green`, with probability $0.4 \cdot 0.9 = 0.36$, and another one uses the probabilistic facts `blue` and `yellow`, with probability $0.5 \cdot 0.6 = 0.30$. Hence, the most likely proof is the first one, represented by the partial world $\{\mathtt{red},\mathtt{green}\}$. However, the MPE is the world $\{\mathtt{green},\mathtt{blue},\mathtt{yellow}\}$, with probability $(1 - 0.4) \cdot 0.9 \cdot 0.5 \cdot 0.6 = 0.162$, which does not extend the partial world $\{\mathtt{red},\mathtt{green}\}$. This counterintuitive result can be seen as a drawback of representing explanations as worlds.

Considering proofs or SLD trees as explanations has obvious advantages: they allows one to follow the chain of inferences from the query back to the considered probabilistic facts and, moreover, can be considered minimal. However, their main weaknesses are their complexity and size, which might be a problem for non-experts.

[8] We only show the relevant bindings of the computed `mgu`'s in the examples.

3.2 Explanations as Programs

In this section, we propose to represent explanations as programs. In principle, we consider that rules and facts are easier to understand than proof trees for non-experts (and could more easily be translated into natural language).[9] Each program thus represents a minimal and more understandable explanation of a proof. Moreover, the generation of explanations is now parametric w.r.t. a set of *visible* predicates, thus hiding unnecessary information. We will then prove that the probability of a query in an explanation is equivalent to that of the associated proof in the original program, and that the marginal probability of a query is preserved when considering the union of all generated explanations.

The explanations of a query are essentially obtained using *unfolding*, a well-known transformation in the area of logic programming [21]. Let $h \leftarrow B, a, B'$ be a clause and $h_1 \leftarrow B_1$,. . . ,$h_n \leftarrow B_n$ be *all* the (renamed apart) clauses whose head unifies with a. Then, unfolding replaces

$$h \leftarrow B, a, B'$$

with the clauses

$$(h \leftarrow B, B_1, B')\theta_1,\ \ \ldots,\ (h \leftarrow B, B_n, B')\theta_n$$

where $\mathsf{mgu}(a, h_i) = \theta_i$, $i = 1, \ldots, n$. In the following, we assume that derived predicates are split into *visible* and *hidden* predicates. In practice, both predicates will be unfolded, but we introduce a special treatment for visible atoms so that their calls are kept in the unfolded clause, and a separate definition is added. Intuitively speaking, visible predicates represent information that the user considers relevant, while hidden predicates represent intermediate or less relevant relations that the user prefers not to see in an explanation.

Given an atom a, we let $\mathit{visible}(a)$ be true if a is rooted by a visible predicate and false otherwise. The list of visible predicates should be given by the user, though a default specification can be considered (e.g., our tool xgen assumes that all predicates are hidden unless otherwise specified).

The generation of explanations is modeled by a number of transition rules. Given a query q, the initial explanation has the form $\{\mathsf{query}(q) \leftarrow q\}$, where we assume that query is a fresh predicate that does not appear in the considered probabilistic program. Then, we aim at unfolding this clause as much as possible. However, there are some relevant differences with the standard unfolding transformation (as in, e.g., [21]):

– First, we do not unfold the clauses of the original program but consider a new program (i.e., the initial explanation). This is sensible in our context since we are only interested in those clauses of the original program that are necessary for proving the query q.

[9] The use of rule-based models to explain the predictions of AI systems is not new in the field of XAI (see, e.g., [3]).

– Second, we keep every nondeterministic unfolding separated in different explanations. This is due to the fact that our explanations represent a *single* proof rather than a complete proof tree.
– Finally, as mentioned above, we distinguish *visible* and *hidden* predicates. While unfolding a hidden predicate follows a standard unfolding, the case of visible predicates is slightly different (see Example 5 below).

During unfolding, we might find four different cases depending on whether the considered clause is probabilistic or not, and whether the considered atom is a derived or a probabilistic atom. In the following, we consider each case separately.

Unfolding of derived atoms in derived clauses. This is the simplest case. Here, unfolding can be performed using the following transition rules, depending on whether the atom is visible or not:

$$\frac{\neg \mathit{visible}(a) \wedge h' \leftarrow B \ll \mathcal{P} \wedge \mathsf{mgu}(a,h') = \theta}{E \cup \{h \leftarrow B_1, a, B_2\} \rightarrowtail (E \cup \{h \leftarrow B_1, B, B_2\})\theta} \quad \text{(unf1)}$$

$$\frac{\mathit{visible}(a) \wedge h' \leftarrow B \ll \mathcal{P} \wedge \mathsf{mgu}(a,h') = \theta \wedge \rho(a\theta) = a'}{E \cup \{h \leftarrow B_1, a, B_2\} \rightarrowtail E\theta \cup \{a' \leftarrow B\theta, h\theta \leftarrow B_1\theta, \underline{a'}, B_2\theta\}} \quad \text{(unf2)}$$

where atoms marked with an underscore (e.g., atom $\underline{a'}$ in rule unf2 above) cannot be selected for unfolding anymore, and ρ is a simple renaming function that takes an atom and returns a new atom with a fresh predicate name and the same arguments (e.g., by adding a suffix to the original predicate name in order to keep its original meaning). For instance, $\rho(\mathtt{smokes(carl)}) = \mathtt{smokes_1(carl)}$. While rule (unf1) denotes a standard unfolding rule (unf2) is a bit more involved. The fact that an atom is visible does not mean that the atom should not be unfolded. It only means that the call should be kept in the unfolded clause in order to preserve the visible components of the inference chain. Indeed, observe that the computed mgu is applied to *all* clauses in the current explanation. This is sensible since all clauses in an explanation actually represent one single proof (i.e., a successful SLD derivation).

Example 5. Consider the following logic program:

```
p(X) :- r(X,Y).        r(X,Y) :- s(Y).        s(b).
```

and the query p(a). A successful SLD derivation for this query is as follows:

$$\mathtt{p(a)} \leadsto \mathtt{r(a,Y)} \leadsto \mathtt{s(Y)} \leadsto_{\{\mathtt{Y}\mapsto\mathtt{b}\}} \mathtt{true}$$

Given the initial explanation $E_0 = \{\mathtt{query(p(a))\ :\!-\ p(a)}\}$, and assuming that no predicate is visible, the (repeated) unfolding of E_0 using rule (unf1) would eventually produce the explanation $E' = \{\mathtt{query(p(a))}\}$, which can be read as "the query q(a) is true". In contrast, if we consider that r/2 is visible, we get the following unfolding sequence:

$E_0 = \{\mathtt{query(p(a))\ :\!-\ p(a)}\}$
$E_1 = \{\mathtt{query(p(a))\ :\!-\ r(a,Y)}\}$ *//using rule* $\mathtt{p(X)\ :\!-\ r(X,Y)}$
$E_2 = \{\mathtt{r_1(a,Y)\ :\!-\ s(Y),\ query(p(a))\ :\!-\ \underline{r_1(a,Y)}}\}$ *//using rule* $\mathtt{r(X,Y)\ :\!-\ s(Y)}$
$E_3 = \{\mathtt{r_1(a,b),\ query(p(a))\ :\!-\ \underline{r_1(a,b)}}\}$ *//using fact* $\mathtt{s(b)}$

The generated explanation (E_3) is a bit more informative than E' above: "the query $\mathtt{p}(\mathtt{a})$ is true because $\mathtt{r}(\mathtt{a},\mathtt{b})$ is true", where $\mathtt{r}/2$ is renamed as $\mathtt{r}_1/2$ in E_3.

In the example above, renaming $\mathtt{r}/2$ is not really needed. However, in general, the renaming of visible atoms is necessary to avoid confusion when unfolding nondeterministic predicates, since we want each explanation to represent one, *and only one*, proof. Consider, e.g., the following program:

```
p :- q,q.    q :- a.   q :- b.   a.   b.
```

ant the query `q`. Assume that predicate `q/0` is visible and that the first call to `q` is unfolded using clause `q :- a` and the second one using clause `q :- b`. Without predicate renaming, we will produce an explanation including a clause of the form $\mathtt{query(p)}$:- $\underline{\mathtt{q}},\underline{\mathtt{q}}$, together with the two clauses defining `q`. Unfortunately, this explanation does not represent a single proof (as intended) since every call to `q` could be unfolded with either clause. Renaming is then needed to ensure that only one unfolding is possible: $\mathtt{query(p)}$:- $\underline{\mathtt{q}_1},\underline{\mathtt{q}_2}$, together with the clauses $\mathtt{q}_1$:- a and $\mathtt{q}_2$:- b.

Unfolding of derived atoms in probabilistic clauses. As mentioned before, probabilistic rules are used to provide an intensional representation of a set of ground probabilistic facts. One could think that the unfolding a derived atom in such a clause will always preserve the probability of a query. However, some caution is required:

Example 6. Consider the following program:

```
q(a,a).   q(a,b).   0.8::p(X) :- q(X,Y).
```

By unfolding clause `0.8::p(X) :- q(X,Y)`, we would get the following program:

```
q(a,a).   q(a,b).   0.8::p(a).   0.8::p(a).
```

Here, $P(\mathtt{p}(\mathtt{a})) = 0.8$ in the first program but $P(\mathtt{p}(\mathtt{a})) = 0.8 \cdot 0.2 + 0.2 \cdot 0.8 + 0.8 \cdot 0.8 = 0.96$ in the second one.

The problem with the above example is related to the interpretation of intensional facts. Observe that *prob_facts* in Definition 1 returns a *set*. This is essential to compute the right probability and to avoid duplicates when there are several successful derivations computing the same ground answer.

Therefore, if we want to preserve the probability of a query, we can only unfold derived atoms when they do not have several proofs computing the same ground answer. Since we are assuming that derivations are finite, this property could be dynamically checked. In the following, for simplicity, we assume instead that programs cannot contain several occurrences of the same (ground) probabilistic fact.[10] The new unfolding rule is thus as follows:

$$\frac{h' \leftarrow B \ll \mathcal{P} \wedge \mathsf{mgu}(a,h') = \theta}{E \cup \{p :: h \leftarrow B_1, a, B_2\} \rightarrowtail (E \cup \{p :: h \leftarrow B_1, B, B_2\})\theta} \quad \text{(unf3)}$$

[10] Nevertheless, our tool xgen considers more general programs by requiring the specification of those predicates that may violate the above condition (see Section 4).

Unfolding of a probabilistic atom in a derived clause. In this case, one might be tempted to define the unfolding of clause $h \leftarrow B_1, a, B_2$ using clause $p :: h' \leftarrow B$ and $\mathsf{mgu}(a, h) = \theta$ as the clause $p :: (h \leftarrow B_1, B, B_2)\theta$. However, such a transformation would generally change the success probability of a query, as illustrated in the following example:

Example 7. Consider the following program

```
    p :- a,b.     p :- b,c.     0.6::a.     0.7::b.     0.8::c.
```

where `p` holds either because `a` and `b` are true or because `b` and `c` are true. By unfolding `b` in the first clause of `p` using the strategy above, we would get

```
0.7::p :- a.      p :- b,c.     0.6::a.     0.7::b.     0.8::c.
```

However, $P(\mathtt{p}) = P(\mathtt{a},\mathtt{b}) + P(\mathtt{b},\mathtt{c}) - P(\mathtt{a},\mathtt{b},\mathtt{c}) = \mathtt{0.6}\cdot\mathtt{0.7} + \mathtt{0.7}\cdot\mathtt{0.8} - \mathtt{0.6}\cdot\mathtt{0.7}\cdot\mathtt{0.8} = \mathtt{0.644}$ in the original program but $P(\mathtt{p}) = 0.7448$ in the unfolded one. Intuitively speaking, the issue is that, by embedding the probability of `b` into the unfolded clause of `p`, the worlds associated with the two proofs of `p` no longer overlap. To be precise, the clause `0.7::p :- a.` is equivalent to `p :- a,pp` with `0.7::pp`. Thus, we now have $P(\mathtt{p}) = P(\mathtt{a},\mathtt{pp}) + P(\mathtt{b},\mathtt{c}) - P(\mathtt{a},\mathtt{pp},\mathtt{b},\mathtt{c}) = \mathtt{0.6}\cdot\mathtt{0.7} + \mathtt{0.7}\cdot\mathtt{0.8} - \mathtt{0.6}\cdot\mathtt{0.7}\cdot\mathtt{0.7}\cdot\mathtt{0.8} = \mathtt{0.7448}$.

Therefore, in the following, probabilistic atoms are always (implicitly) considered as *visible* atoms:

$$\frac{p :: h' \leftarrow B \ll \mathcal{P} \wedge \mathsf{mgu}(a, h') = \theta}{E \cup \{h \leftarrow B_1, a, B_2\} \rightarrowtail (E \cup \{p :: h' \leftarrow B, h \leftarrow B_1, \underline{a}, B_2\})\theta} \quad \text{(unf4)}$$

Note that the probabilistic atom is not renamed, in contrast to the renaming of visible atoms in rule (unf2) above. Renaming would be required only if a program could have several probabilistic atoms with different probabilities, thus introducing some undesired nondeterminism (but we ruled out this possibility, as mentioned before).

Unfolding of a probabilistic atom in a probabilistic rule. Although this situation cannot happen in the original program (we required intensional facts to have only derived atoms in their bodies), such a situation may show up after a number of unfolding steps. This case is similar to unfolding a probabilistic atom in a derived clause:

$$\frac{p' :: h' \leftarrow B \ll \mathcal{P} \wedge \mathsf{mgu}(a, h') = \theta}{E \cup \{p :: h \leftarrow B_1, a, B_2\} \rightarrowtail (E \cup \{p' :: h' \leftarrow B, p :: h \leftarrow B_1, \underline{a}, B_2\})\theta} \quad \text{(unf5)}$$

In the following, given some initial explanation E_0, we refer to a sequence $E_0 \rightarrowtail E_1 \rightarrowtail \ldots \rightarrowtail E_n$, $n \geq 0$, as an *unfolding sequence.* If further unfolding steps are possible, we say that E_n is a *partial explanation.* Otherwise, if $E_n \not\rightarrowtail$, we have two possibilities:

– If the clauses in E_n contain no selectable atom (i.e., all atoms in the bodies of the clauses are either *true* or have the form $\underline{a}$), then E_n is called a *successful explanation* and we refer to $E_0 \rightarrowtail \ldots \rightarrowtail E_n$ as a *successful unfolding sequence*. The *probability of a successful explanation*, $P(E_n)$, can be simply obtained as the product of the probabilities of the probabilistic clauses in this explanation.
– If the clauses in E_n contain some selectable atom which does not unify with the head of any program clause, then E_n is called a *failing explanation* and it is discarded from the generation process.

By construction, there exists one successful unfolding sequence associated with each successful SLD derivation of a query.

Example 8. Consider again the program in Example 1, where we now add one additional ground probabilistic fact: `0.1::influences(ann,bob)`. Moreover, assume that predicate `smokes/1` is visible. Then, we have the following successful explanation sequence:

$$\begin{aligned}
E_0 &= \{\ \mathsf{query}(\mathtt{smokes}(\mathtt{carl}))\ \mathtt{:-}\ \mathtt{smokes}(\mathtt{carl}) \ \}\\
E_1 &= \{\ \mathsf{query}(\mathtt{smokes}(\mathtt{carl}))\ \mathtt{:-}\ \mathtt{influences}(\mathtt{bob},\mathtt{carl}), \mathtt{smokes}(\mathtt{bob}) \ \}\\
E_2 &= \{\ \mathsf{query}(\mathtt{smokes}(\mathtt{carl}))\ \mathtt{:-}\ \underline{\mathtt{influences}(\mathtt{bob},\mathtt{carl})}, \mathtt{smokes}(\mathtt{bob}),\\
&\qquad \mathsf{0.3} :: \mathtt{influences}(\mathtt{bob},\mathtt{carl}) \ \}\\
E_3 &= \{\ \mathsf{query}(\mathtt{smokes}(\mathtt{carl}))\ \mathtt{:-}\ \underline{\mathtt{influences}(\mathtt{bob},\mathtt{carl})}, \underline{\mathtt{smokes}_1(\mathtt{bob})},\\
&\qquad \mathsf{0.3} :: \mathtt{influences}(\mathtt{bob},\mathtt{carl}),\quad \mathtt{smokes}_1(\mathtt{bob})\ \mathtt{:-}\ \mathtt{stress}(\mathtt{bob}) \ \}\\
E_4 &= \{\ \mathsf{query}(\mathtt{smokes}(\mathtt{carl}))\ \mathtt{:-}\ \underline{\mathtt{influences}(\mathtt{bob},\mathtt{carl})}, \underline{\mathtt{smokes}_1(\mathtt{bob})},\\
&\qquad \mathsf{0.3} :: \mathtt{influences}(\mathtt{bob},\mathtt{carl}),\quad \mathtt{smokes}_1(\mathtt{bob})\ \mathtt{:-}\ \underline{\mathtt{stress}(\mathtt{bob})},\\
&\qquad \mathsf{0.8} :: \mathtt{stress}(\mathtt{bob}) \ \}
\end{aligned}$$

Therefore, E_4 is a successful explanation for the query with probability $P(E_4) = \mathsf{0.24}$, and can be read as "`carl` smokes because `bob` influences `carl` (with probability 0.3) and `bob` smokes, and `bob` smokes because he is stressed (with probability 0.8)". There exists another (less likely) explanation E' as follows:

$$\begin{aligned}
E' = \{\ &\mathtt{smokes}(\mathtt{carl})\ \text{:-}\ \underline{\mathtt{influences}(\mathtt{bob},\mathtt{carl})}, \underline{\mathtt{smokes}_1(\mathtt{bob})},\\
&\mathtt{smokes}_1(\mathtt{bob})\ \text{:-}\ \underline{\mathtt{influences}(\mathtt{ann},\mathtt{bob})}, \underline{\mathtt{smokes}_2(\mathtt{ann})},\\
&\mathtt{smokes}_2(\mathtt{ann})\ \text{:-}\ \underline{\mathtt{stress}(\mathtt{ann})},\quad \mathsf{0.1} :: \mathtt{influences}(\mathtt{ann},\mathtt{bob}),\\
&\mathsf{0.3} :: \mathtt{influences}(\mathtt{bob},\mathtt{carl}),\quad \mathsf{0.8} :: \mathtt{stress}(\mathtt{ann}) \ \}
\end{aligned}$$

with probability $P(E') = \mathsf{0.024}$. Note that the probability of the query in $E_4 \cup E'$ is the same as in the original program: 0.2448 (and different from $P(E_4)+P(E')$).

Correctness. Our main result states the soundness and completeness of successful explanations:[11]

[11] Proofs of technical results can be found in [35].

Theorem 1. *Let $\mathcal{P}$ be a program and q a query. Then, q has a successful SLD derivation in $\mathcal{P}$ with (ground) computed answer θ iff there exists a successful explanation E such that* $\mathsf{query}(q)$ *has one, and only one, successful SLD derivation in E computing the same answer θ and using the same probabilistic clauses.*

As a consequence, we have the following property that states that the probability of a successful derivation can be obtained from the product of the probabilistic clauses in the associated explanation:

Corollary 1. *Let $\mathcal{P}$ be a program and q a query. Then, there is a successful derivation D for q in $\mathcal{P}$ iff there is a successful explanation E with $P(D) = P(E)$. Moreover, $P(E) = P(\mathsf{query}(q))$ in E.*

The above result is an easy consequence of Theorem 1 and the fact that E contains all, and only, the probabilistic facts required for the considered derivation. Note that the success probability of a query and that of a proof trivially coincide in successful explanations since only one proof per explanation exists.

Finally, we consider the preservation of the success probability of a query in the union of generated explanations:

Theorem 2. *Let $\mathcal{P}$ be a program and q a query. Let $E_1, \ldots, E_n$ be all an only the successful explanations for q in $\mathcal{P}$. Then, $P(q)$ in $\mathcal{P}$ is equal to $P(\mathsf{query}(q))$ in $E_1 \cup \ldots \cup E_n$, $n \geq 0$.*

4 The Explanation Generator xgen

In order to put into practice the ideas introduced so far, we have developed a proof-of-concept implementation of the explanation generator, called xgen. The tool has been implemented in SWI Prolog and includes four modules and approximately one thousand lines of code. The main module implementing the transition rules of the previous section has some 300 lines of Prolog code. This module also implements an unfolding *strategy* that ensures termination in many cases (see the discussion below). The remaining modules implement some utility predicates as well as the parser of ProbLog files with visibility annotations. The tool can be downloaded from `https://github.com/mistupv/xgen`.

The tool accepts ProbLog programs containing probabilistic facts defined by (not necessarily ground) facts and rules (i.e., intensional facts). The user can also (optionally) specify which predicates are *visible* (if any) by means of annotations. Furthermore, xgen accepts duplicated probabilistic facts as long as the corresponding predicates are declared *unsafe* using an annotation. Unsafe atoms are dealt with similarly to visible atoms when they occur in probabilistic clauses, but can be unfolded freely when they appear in a derived clause (this is why a new annotation is required). An example specifying an unsafe predicate can be found in the above URL.

As in ProbLog, a query q is added to the program as a fact of the form $\mathsf{query}(q)$. Let us consider, for instance, the program from Example 8, where we

```
$ swipl
Welcome to SWI-Prolog (version 8.2.4)
[...]
?- [xgen].
true.
?- xgen('examples/smokes_paper.pl').
% Explanation #1:
0.3::infl(bob,carl).
0.8::stress(bob).
smokes(bob) :- stress(bob).
smokes(carl) :- infl(bob,carl),smokes(bob).
query(smokes(carl)).
% Success probability: 0.24

% Explanation #2:
0.1::infl(ann,bob).
0.3::infl(bob,carl).
0.8::stress(ann).
smokes(bob) :- infl(ann,bob),smokes0(ann).
smokes(carl) :- infl(bob,carl),smokes(bob).
smokes0(ann) :- stress(ann).
query(smokes(carl)).
% Success probability: 0.024

% No more explanations...

% Combined explanations:
0.1::infl(ann,bob).
0.3::infl(bob,carl).
0.8::stress(ann).
0.8::stress(bob).
smokes(bob) :- stress(bob).
smokes(bob) :- infl(ann,bob),smokes0(ann).
smokes(carl) :- infl(bob,carl),smokes(bob).
smokes0(ann) :- stress(ann).
query(smokes(carl)).
% Success probability: 0.2448

Output files can be found in folder
"explanations".
```

Fig. 2. A typical session with xgen

now add the query as the fact `query(smokes(carl))`. If the program is stored in file `smokes_paper.pl`, a typical session proceeds as shown in Figure 2, where the predicate `influences/2` is abbreviated to `infl/2`.

In order to deal with cyclic definitions, we have implemented the following unfolding strategy in xgen: we select the leftmost atom in the body of a clause that is not underlined nor a *variant* of any of its (instantiated) *ancestors*.[12] For instance, our tool can deal with programs containing cyclic definitions like

```
path(X,Y) :- edge(X,Y).
path(X,Y) :- path(Z,Y), edge(X,Z).
```

(together with a set of probabilistic facts defining `edge/2`). Similar strategies have been used, e.g., in partial deduction [5]. Although this strategy is clearly sound—an infinite derivation must necessarily select the same atom once and again, since the Herbrand universe is finite—completeness does not generally hold (a counterexample can be found in [6]). As a consequence, our unfolding strategy could prune some derivations despite the fact that they can eventually give rise to a successful SLD derivation. Nevertheless, completeness can still be guaranteed for certain classes of programs, like the *restricted* programs of [6] that, intuitively speaking, only allow one recursive call in the bodies of recursive predicates (as in the definition of predicate `path/2` above), so that no infinitely growing queries can be obtained. The class of restricted programs is similar to that of *B-stratifiable* logic programs [12] and its generalization, *strongly regular* logic programs [34], both used in the context of partial deduction [16]. As an alternative to using a terminating unfolding strategy as we do in xgen, one could

[12] Let $B_1, a, B_2 \leadsto_\theta (B_1, B, B_2)\theta$ an SLD resolution step using clause $h \leftarrow B$ and $\mathsf{mgu}(a, h) = \theta$. Then, a is the direct ancestor of the atoms in B. The notion of ancestor is the transitive closure of this relation.

also consider an implementation of *tabled* SLD resolution (as in [11]) or an iterative deepening strategy (as in [17]). Nevertheless, termination is not decidable for general logic programs no matter the considered strategy.

5 Related Work

An obvious related work is the definition of fold/unfold transformations for logic programs (see [21] and references herein). Indeed, rules (unf1) and (unf3) can be seen as standard unfolding transformations (except for the differences already mentioned in Section 3.2). In general, given a program $\mathcal{P}$ and an initial explanation $E_0 = \{\mathsf{query}(q) \leftarrow q\}$, a standard unfolding transformation on $\mathcal{P} \cup E_0$ would return $\mathcal{P} \cup E_1 \cup \ldots \cup E_n$, where $E_0 \rightarrowtail E_1, \ldots, E_0 \rightarrowtail E_n$ are all the possible unfolding steps using rules (unf1) and (unf3). Rules for visible or probabilistic atoms, i.e., rules (unf2), (unf4) and (unf5), resemble a combination of definition introduction and folding, followed by unfolding. Nevertheless, a distinctive feature of our approach is that the computed bindings are shared by all the clauses in the explanation. Indeed, applying the computed mgu's to *all* clauses in the current explanation is sensible in our context since all clauses together represent a *single* proof.

To the best of our knowledge, the only previous approach to defining an unfolding transformation in the context of a probabilistic logic formalism is that in [20]. However, this work considers *stochastic logic programs* (SLPs) [19], a generalization of stochastic grammars and hidden Markov models. SLPs do not follow the distribution semantics (as PLP does). Actually, the unfolding transformation in [20] is the standard one for logic programs [21]. Here, the probability is always preserved by the unfolding transformation because of the way the probability of SLPs is computed (i.e., the probability of a query is obtained directly from the successful SLD derivations of the query).

In the context of PLP with a distribution semantics, we are not aware of any previous work focused on unfolding transformations or on the generation of explanations other than computing the MPE [31] or Viturbi proof [14]. Actually, we find more similarities between our approach and the technique called *knowledge-based model construction* [13] used to compute the grounding of the program clauses which are relevant for a given query (see also [11]). However, both the aim and the technique are different from ours.

Finally, let us mention some recent advances to improve the quality of explanations in a closely related field: *Answer Set Programming* (ASP) [7]. First, [8] presents a tool, `xclingo`, for generating explanations from annotated ASP programs. Annotations are then used to construct derivation trees containing textual explanations. Moreover, the language allows the user to select *which* atoms or rules should be included in the explanations. And, second, [2] presents so-called *justifications* for ASP programs with constraints, now based on a goal-directed semantics. As in the previous work, the user can decide the level of detail required in a justification tree, as well as add annotations to produce justifications using natural language. Obviously, our work shares the aim of these papers regarding the generation of minimal and understandable explanations. However, the considered language and the applied techniques are different. Nevertheless, we find it very interesting to extend our work with some of the ideas in [2,8], e.g., the use of annotations to produce explanations using natural language.

6 Concluding Remarks and Future Work

In this paper, we have presented a novel approach to generate explanations in the context of PLP languages like ProbLog [25]. In particular, and in contrast to previous approaches, we have proposed explanations to be represented as programs, one for each proof of a given query. In this way, the user can analyze each (minimal) proof separately, understand *why* the considered prediction (query) is true following the chain of inferences (an intuitive process) and using a familiar control structure, that of conditional rules. We have formally proved that explanations preserve the probability of the original proofs, and that the success probability of a query can also be computed from the union of the generated explanations. A proof-of-concept tool for generating explanations, `xgen`, has been implemented, demonstrating the viability of the approach.

We consider several avenues for future work. On the one hand, we plan to extend the features of the considered language in order to include negation,[13] disjunctive probabilistic clauses, evidences, some Prolog built-in's, etc. This will surely improve the applicability of our approach and will allow us to carry on an experimental evaluation of the technique. In particular, we plan to study both the scalability of the approach as well as the usefulness of the generated explanations w.r.t. some selected case studies.

Another interesting research line consists in allowing the addition of annotations in program clauses so that natural language explanations can be generated (as in [2,8]). Finally, we would also like to explore the use of our unfolding transformation as a pre-processing stage for computing the probability of a query. In particular, when no predicate is declared as visible, our transformation produces a number of explanations of the form $\{p_1 :: a_1, \ldots,\ p_n :: a_n,\ \mathsf{query}(q) \leftarrow \underline{a_1}, \ldots, \underline{a_n}\}$, where $p_1 :: a_1, \ldots, p_n :: a_n$ are ground probabilistic facts. Apparently, computing the probability of a query from the union of the generated explanations seems much simpler than computing it for an arbitrary program.

Acknowledgements. I would like to thank the anonymous reviewers for their suggestions to improve this paper.

References

1. Apt, K.R.: From Logic Programming to Prolog. Prentice Hall (1997)

[13] Considering negated *ground* probabilistic facts is straightforward, but dealing with negated derived atoms is much more challenging.

2. Arias, J., Carro, M., Chen, Z., Gupta, G.: Justifications for goal-directed constraint answer set programming. In: Ricca, F., Russo, A., Greco, S., Leone, N., Artikis, A., Friedrich, G., Fodor, P., Kimmig, A., Lisi, F.A., Maratea, M., Mileo, A., Riguzzi, F. (eds.) Proceedings of the 36th International Conference on Logic Programming (ICLP Technical Communications 2020). EPTCS, vol. 325, pp. 59–72 (2020). https://doi.org/10.4204/EPTCS.325.12
3. Arrieta, A.B., Rodríguez, N.D., Ser, J.D., Bennetot, A., Tabik, S., Barbado, A., García, S., Gil-Lopez, S., Molina, D., Benjamins, R., Chatila, R., Herrera, F.: Explainable artificial intelligence (XAI): concepts, taxonomies, opportunities and challenges toward responsible AI. Inf. Fusion **58**, 82–115 (2020). https://doi.org/10.1016/j.inffus.2019.12.012
4. Azzolini, D., Riguzzi, F.: Syntactic Requirements for Well-defined Hybrid Probabilistic Logic Programs. In: Formisano, A., Liu, Y.A., Bogaerts, B., Brik, A., Dahl, V., Dodaro, C., Fodor, P., Pozzato, G.L., Vennekens, J., Zhou, N. (eds.) Proceedings of the 37th International Conference on Logic Programming (Technical Communications) (ICLP Technical Communications 2021). EPTCS, vol. 345, pp. 14–26 (2021). https://doi.org/10.4204/EPTCS.345.12, `https://doi.org/10.4204/EPTCS.345.12`
5. Bol, R.N.: Loop checking in partial deduction. J. Log. Program. **16**(1), 25–46 (1993). https://doi.org/10.1016/0743-1066(93)90022-9
6. Bol, R.N., Apt, K.R., Klop, J.W.: An analysis of loop checking mechanisms for logic programs. Theor. Comput. Sci. **86**(1), 35–79 (1991). https://doi.org/10.1016/0304-3975(91)90004-L, `https://doi.org/10.1016/0304-3975(91)90004-L`
7. Brewka, G., Eiter, T., Truszczynski, M.: Answer set programming at a glance. Commun. ACM **54**(12), 92–103 (2011). https://doi.org/10.1145/2043174.2043195
8. Cabalar, P., Fandinno, J., Muñiz, B.: A system for explainable answer set programming. In: Ricca, F., Russo, A., Greco, S., Leone, N., Artikis, A., Friedrich, G., Fodor, P., Kimmig, A., Lisi, F.A., Maratea, M., Mileo, A., Riguzzi, F. (eds.) Proceedings of the 36th International Conference on Logic Programming (ICLP Technical Communications 2020). EPTCS, vol. 325, pp. 124–136 (2020). https://doi.org/10.4204/EPTCS.325.19
9. Choudhury, A., Gupta, D.: A survey on medical diagnosis of diabetes using machine learning techniques. In: Kalita, J., Balas, V.E., Borah, S., Pradhan, R. (eds.) Recent Developments in Machine Learning and Data Analytics. pp. 67–78. Springer Singapore, Singapore (2019)
10. EU, EEA: Regulation (EU) 2016/679 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, available from `https://eur-lex.europa.eu/eli/reg/2016/679/oj`
11. Fierens, D., den Broeck, G.V., Renkens, J., Shterionov, D.S., Gutmann, B., Thon, I., Janssens, G., Raedt, L.D.: Inference and learning in probabilistic logic programs using weighted boolean formulas. Theory Pract. Log. Program. **15**(3), 358–401 (2015). https://doi.org/10.1017/S1471068414000076
12. Hruza, J., Stepánek, P.: Speedup of logic programs by binarization and partial deduction. Theory Pract. Log. Program. **4**(3), 355–380 (2004). https://doi.org/10.1017/S147106840300190X
13. Kersting, K., Raedt, L.D.: Bayesian logic programs. CoRR **cs.AI/0111058** (2001), `https://arxiv.org/abs/cs/0111058`
14. Kimmig, A., Demoen, B., Raedt, L.D., Costa, V.S., Rocha, R.: On the implementation of the probabilistic logic programming language ProbLog. Theory Pract. Log. Program. **11**(2-3), 235–262 (2011). https://doi.org/10.1017/S1471068410000566

15. Lloyd, J.W.: Foundations of Logic Programming, 2nd Edition. Springer (1987). https://doi.org/10.1007/978-3-642-83189-8
16. Lloyd, J.W., Shepherdson, J.C.: Partial evaluation in logic programming. J. Log. Program. **11**(3&4), 217–242 (1991). https://doi.org/10.1016/0743-1066(91)90027-M
17. Mantadelis, T., Rocha, R.: Using iterative deepening for probabilistic logic inference. In: Lierler, Y., Taha, W. (eds.) Proceedings of the 19th International Symposium Practical Aspects of Declarative Languages (PADL 2017). Lecture Notes in Computer Science, vol. 10137, pp. 198–213. Springer (2017). https://doi.org/10.1007/978-3-319-51676-9_14
18. Molnar, C., Casalicchio, G., Bischl, B.: Interpretable machine learning - A brief history, state-of-the-art and challenges. In: Proceedings of the Second International Workshop on eXplainable Knowledge Discovery in Data Mining (XKDD 2020). Communications in Computer and Information Science, vol. 1323, pp. 417–431. Springer (2020). https://doi.org/10.1007/978-3-030-65965-3_28
19. Muggleton, S.: Stochastic logic programs. In: de Raedt, L. (ed.) Advances in Inductive Logic Programming, pp. 254–264. IOS Press (1996)
20. Muggleton, S.: Semantics and derivation for stochastic logic programs. In: Proceedings of the UAI-2000 Workshop on Fusion of Domain Knowledge with Data for Decision Support (2000)
21. Pettorossi, A., Proietti, M.: Transformation of logic programs: Foundations and techniques. J. Log. Program. **19/20**, 261–320 (1994). https://doi.org/10.1016/0743-1066(94)90028-0
22. Poole, D.: Probabilistic horn abduction and bayesian networks. Artif. Intell. **64**(1), 81–129 (1993). https://doi.org/10.1016/0004-3702(93)90061-F
23. Poole, D.: The independent choice logic for modelling multiple agents under uncertainty. Artif. Intell. **94**(1-2), 7–56 (1997). https://doi.org/10.1016/S0004-3702(97)00027-1
24. Raedt, L.D., Kimmig, A.: Probabilistic (logic) programming concepts. Mach. Learn. **100**(1), 5–47 (2015). https://doi.org/10.1007/s10994-015-5494-z
25. Raedt, L.D., Kimmig, A., Toivonen, H.: ProbLog: A probabilistic Prolog and its application in link discovery. In: Veloso, M.M. (ed.) Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007). pp. 2462–2467 (2007), `http://ijcai.org/Proceedings/07/Papers/396.pdf`
26. Ricci, F., Rokach, L., Shapira, B. (eds.): Recommender Systems Handbook. Springer (2015). https://doi.org/10.1007/978-1-4899-7637-6
27. Riguzzi, F.: Foundations of Probabilistic Logic Programming: Languages, Semantics, Inference and Learning. River Publishers (2018)
28. Riguzzi, F., Swift, T.: Well-definedness and efficient inference for probabilistic logic programming under the distribution semantics. Theory Pract. Log. Program. **13**(2), 279–302 (2013). https://doi.org/10.1017/S1471068411000664, `https://doi.org/10.1017/S1471068411000664`
29. Sato, T.: A statistical learning method for logic programs with distribution semantics. In: Sterling, L. (ed.) Logic Programming, Proceedings of the Twelfth International Conference on Logic Programming, Tokyo, Japan, June 13-16, 1995. pp. 715–729. MIT Press (1995)
30. Sato, T., Kameya, Y.: PRISM: A language for symbolic-statistical modeling. In: Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI 97, Nagoya, Japan, August 23-29, 1997, 2 Volumes. pp. 1330–1339. Morgan Kaufmann (1997), `http://ijcai.org/Proceedings/97-2/Papers/078.pdf`

31. Shterionov, D.S., Renkens, J., Vlasselaer, J., Kimmig, A., Meert, W., Janssens, G.: The most probable explanation for probabilistic logic programs with annotated disjunctions. In: Davis, J., Ramon, J. (eds.) Proceedings of the 24th International Conference on Inductive Logic Programming (ILP 2014). Lecture Notes in Computer Science, vol. 9046, pp. 139–153. Springer (2014). https://doi.org/10.1007/978-3-319-23708-4_10
32. Vennekens, J., Verbaeten, S., Bruynooghe, M.: Logic programs with annotated disjunctions. In: Demoen, B., Lifschitz, V. (eds.) Logic Programming, 20th International Conference, ICLP 2004, Saint-Malo, France, September 6-10, 2004, Proceedings. Lecture Notes in Computer Science, vol. 3132, pp. 431–445. Springer (2004). https://doi.org/10.1007/978-3-540-27775-0_30
33. Veres, M., Moussa, M.: Deep learning for intelligent transportation systems: A survey of emerging trends. IEEE Transactions on Intelligent Transportation Systems **21**(8), 3152–3168 (2020). https://doi.org/10.1109/TITS.2019.2929020
34. Vidal, G.: A hybrid approach to conjunctive partial evaluation of logic programs. In: Alpuente, M. (ed.) Proceedings of the 20th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2010), Revised Selected Papers. Lecture Notes in Computer Science, vol. 6564, pp. 200–214. Springer (2010). https://doi.org/10.1007/978-3-642-20551-4_13
35. Vidal, G.: Explanations as Programs in Probabilistic Logic Programming (2022), available from `http://personales.upv.es/gvidal/german/flops22/tr.pdf`