Logical Methods in Computer Science
Volume 17, Issue 2, 2021, pp. 2:1–2:35
https://lmcs.episciences.org/
Submitted Mar. 07, 2020
Published Apr. 12, 2021

COALGEBRAIC SEMANTICS FOR PROBABILISTIC LOGIC PROGRAMMING

TAO GU AND FABIO ZANASI

University College London
e-mail address: tao.gu.18@ucl.ac.uk
e-mail address: f.zanasi@ucl.ac.uk

Abstract. Probabilistic logic programming is increasingly important in artificial intelligence and related fields as a formalism to reason about uncertainty. It generalises logic programming with the possibility of annotating clauses with probabilities. This paper proposes a coalgebraic semantics on probabilistic logic programming. Programs are modelled as coalgebras for a certain functor F, and two semantics are given in terms of cofree coalgebras. First, the cofree F-coalgebra yields a semantics in terms of derivation trees. Second, by embedding F into another type G, as cofree G-coalgebra we obtain a 'possible worlds' interpretation of programs, from which one may recover the usual distribution semantics of probabilistic logic programming. Furthermore, we show that a similar approach can be used to provide a coalgebraic semantics to weighted logic programming.

1. Introduction

Probabilistic logic programming (PLP) [NS92, Dan92, Sat95] is a family of approaches extending the declarative paradigm of logic programming with the possibility of reasoning about uncertainty. This has been proven useful in various applications, including bioinformatics [DRKT07a, MH12], robotics [TBF05] and the semantic web [Zes17].

The most common version of PLP — on which for instance ProbLog is based [DRKT07a], the probabilistic analogue of Prolog — is defined by annotating clauses in programs with mutually independent probabilities. As for the interpretation, *distribution semantics* [Sat95] is typically used as a benchmark for the various implementations of PLP, such as pD, PRISM and ProbLog [RS14]. While in logic programming the central task is whether a goal is provable using the clauses in the program as axioms, for PLP the typical question one asks is what is the probability of a goal being provable. In distribution semantics, all the clauses are seen as independent random events, and such probability is obtained as the sum of the probabilities of all the *possible worlds* (sets of clauses) in which the goal is provable. The distribution semantics is particularly interesting because it is compatible with the encoding of Bayesian networks as probabilistic logic programs [RS14], thus indicating that PLP can be effectively employed for Bayesian reasoning.

Key words and phrases: probabilistic logic programming, distribution semantics, weighted logic programming.

DOI:10.23638/LMCS-17(2:2)2021
© T. Gu and F. Zanasi
Creative Commons

The main goal of this work is to present a coalgebraic perspective on PLP and its distribution semantics. We first consider the case of ground programs (Section 3), that is, programs without variables. Our approach is based on the observation — inspired by the coalgebraic treatment of 'pure' logic programming [KMP10] — that ground programs are in 1-1 correspondence with coalgebras for the functor $\mathcal{M}_{pr}\mathcal{P}_f$, where $\mathcal{M}_{pr}$ is the finite multiset functor on $[0,1]$ and $\mathcal{P}_f$ is the finite powerset functor. We then provide two coalgebraic semantics for ground PLP.

- The first interpretation $[\![-]\!]$ is in terms of execution trees called *stochastic derivation trees*, which represent parallel SLD-derivations of a program on a goal. Stochastic derivation trees are the elements of the cofree $\mathcal{M}_{pr}\mathcal{P}_f$-coalgebra on a given set of atoms At, meaning that any goal $A \in \mathsf{At}$ can be given a semantics in terms of the corresponding stochastic derivation tree by the universal map $[\![-]\!]$ to the cofree coalgebra.
- The second interpretation $\langle\!\langle - \rangle\!\rangle$ recovers the usual distribution semantics of PLP. This requires some work, as expressing probability distributions on the possible worlds needs a different coalgebra type. We introduce *distribution trees*, a tree-like representation of the distribution semantics, as the elements of the cofree $\mathcal{D}_{\leq 1}\mathcal{P}_f\mathcal{P}_f$-coalgebra on At, where $\mathcal{D}_{\leq 1}$ is the sub-probability distribution monad. In order to characterise $\langle\!\langle - \rangle\!\rangle$ as the map given by universal property of distribution trees, we need a canonical extension of PLP to the setting of $\mathcal{D}_{\leq 1}\mathcal{P}_f\mathcal{P}_f$-coalgebras. This is achieved via a 'possible worlds' natural transformation $\mathcal{M}_{pr}\mathcal{P}_f \Rightarrow \mathcal{D}_{\leq 1}\mathcal{P}_f\mathcal{P}_f$.

In the second part of the paper (Section 4) we recover the same framework for arbitrary probabilistic logic programs, possibly including variables. The encoding of programs as coalgebras is subtler. The space of atoms is now a presheaf indexed by a 'Lawvere theory' of terms and substitutions. The coalgebra map can be defined in different ways, depending on the substitution mechanism on which one wants to base resolution. For pure logic programs, the definition by term-matching is the best studied, with [KP11] observing that moving from sets to posets is required in order for the corresponding coalgebra map to be well-defined as a natural transformation between presheaves. A different route is taken in [BZ15], where the problem of naturality is neutralised via 'saturation', a categorical construction which amounts to defining resolution by unification instead of term-matching.

In developing a coalgebraic treatment of PLP with variables, we follow the saturation route, as it also allows to recover the term-matching approach, via 'desaturation' [BZ15]. This provides a cofree coalgebra semantics $[\![-]\!]$ for arbitrary PLP programs, as a rather straightforward generalisation of the saturated semantics for pure logic programs. On the other hand, extending the ground distribution semantics $\langle\!\langle - \rangle\!\rangle$ to arbitrary PLP programs poses some challenges: we need to ensure that, in computing the distribution over possible worlds associated to each sub-goal in the computation, each clause of the program is 'counted' only once. This is solved by tweaking the coalgebra type of the distribution trees for arbitrary PLP programs, so that some nodes are labelled with clauses of the program. Thanks to this additional information, the term-matching distribution semantics of an arbitrary PLP goal is computable from its distribution tree.

After developing our framework for PLP, in the last part of the paper (Section 5) we show how the same approach yields a coalgebraic semantics for weighted logic programming (WLP). WLP generalises standard logic programming by adding weights to clauses; it is mostly used in the specification of dynamic programming algorithms in various fields, including natural language processing [EB20] and computational biology [DEKM98]. Our coalgebraic

description yields a derivation semantics $[\![-]\!]$ both for ground and arbitrary WLP programs, from which one may compute the weight associated with a goal.

In light of the coalgebraic treatment of pure logic programming [KMP10, KP11, BZ13, BZ15], the generalisation to PLP and WLP may not appear so surprising. In fact, we believe its importance is two-fold. First, whereas the derivation semantics $[\![-]\!]$ is a straight generalisation of the pure setting, the distribution semantics $\langle\!\langle - \rangle\!\rangle$ is genuinely novel, and does not have counterparts in pure logic programming. Second, this work provides a starting point for a generic coalgebraic treatment of variations of logic programming:

- The coalgebraic approach to pure logic programming has been used as a formal justification [KPS16, KL17] for coinductive logic programming [KL18, GBM$^+$07]. Coinduction in the context of probabilistic and weighted logic programs is, to the best of our knowledge, a completely unexplored field, for which the current paper establishes semantic foundations.
- As mentioned, reasoning in Bayesian networks can be seen as a particular case of PLP, equipped with the distribution semantics. Our coalgebraic perspective thus readily applies to Bayesian reasoning, paving the way for combination with recent works [JKZ19, JZ19, DDGK16] modelling belief revision, causal inference and other Bayesian tasks in algebraic terms.

We leave the exploration of these venues as follow-up work.

This work extends the conference paper [GZ19] with the addition of the missing proofs (Appendix D), and novel material on weighted logic programming (Section 5 and Appendix C).

2. Preliminaries

Signature, Terms, and Categories. A *signature* Σ is a set of function symbols, each equipped with a fixed finite arity. Throughout this paper we fix a signature Σ, and a countably infinite set of variables $\mathit{Var} = \{x_1, x_2, \dots\}$. The Σ-terms over Var are defined as usual. A *context* is a finite sequence of variables $\langle x_1, x_2, \dots, x_n\rangle$. With some abuse of notation, we shall often use n to denote this context. We say a Σ-term t is *compatible* with context n if the variables appearing in t are all contained in $\{x_1, \dots, x_n\}$.

We are going to reason about Σ-terms categorically using Lawvere theories. First, we will use $\mathsf{Ob}(\mathbf{C})$ to denote the set of objects and $\mathbf{C}[C, D]$ for the set of morphisms $C \to D$ in a category $\mathbf{C}$. A $\mathbf{C}$-indexed *presheaf* is a functor $\mathsf{F}\colon \mathbf{C} \to \mathbf{Sets}$. $\mathbf{C}$-indexed presheaves and natural transformations between them form a category $\mathbf{Sets}^{\mathbf{C}}$. Recall that the (opposite) Lawvere Theory of Σ is the category $\mathbf{L}_\Sigma^{\mathrm{op}}$ with objects the natural numbers and morphisms $\mathbf{L}_\Sigma^{\mathrm{op}}[n, m]$ the n-tuples $\langle t_1, \dots, t_n\rangle$, where each t_i is a Σ-term in context m. For modelling logic programming, it is convenient to think of each $n \in \mathsf{Ob}(\mathbf{L}_\Sigma^{\mathrm{op}})$ as representing the context $\langle x_1, \dots, x_n\rangle$, and a morphism $\langle t_1, \dots, t_n\rangle : n \to m$ as the substitution transforming Σ-terms in context n to Σ-terms in context m by replacing each x_i with t_i. For this reason we shall also refer to $\mathbf{L}_\Sigma^{\mathrm{op}}$ morphisms simply as substitutions (notation $\theta, \tau, \sigma, \dots$).

Logic programming. We now recall the basics of (pure) logic programming, and refer the reader to [Llo87] for a more systematic exposition. An *alphabet* $\mathcal{A}$ consists of a signature Σ, a set of variables Var, and a set of predicate symbols $\{P_1, P_2, \dots\}$, each with a fixed finite arity. Given an n-ary predicate symbol P in $\mathcal{A}$, and Σ-terms $t_1, \dots, t_n$, $P(t_1 \cdots t_n)$ is called an *atom* over $\mathcal{A}$. We use $A, B, \dots$ to denote atoms. Given an atom A in context n, and a substitution $\theta = \langle t_1, \dots, t_n\rangle : n \to m$, we write $A\theta$ for the *substitution instance* of A obtained by replacing each appearance of x_i with t_i in A. For convenience, we also use

$\{B_1, \ldots, B_k\}\theta$ as a shorthand for $\{B_1\theta, \ldots, B_k\theta\}$. Given two atoms A and B (over $\mathcal{A}$), a *unifier* of A and B is a pair $\langle \sigma, \tau \rangle$ of substitutions such that $A\sigma = B\tau$. *Term matching* is a special case of unification, where σ is the identity substitution. In this case we say that τ matches B with A if $A = B\tau$.

A (pure) logic program $\mathbb{L}$ consists of a finite set of clauses $\mathcal{C}$ in the form $H \leftarrow B_1, \ldots, B_k$, where $H, B_1, \ldots, B_k$ are atoms. H is called the *head* of $\mathcal{C}$, and $B_1, \ldots, B_k$ form the *body* of $\mathcal{C}$. We denote H by $\mathsf{Head}(\mathcal{C})$, and $\{B_1, \ldots, B_k\}$ by $\mathsf{Body}(\mathcal{C})$. A clause $\mathcal{C}$ with empty body is also called a *fact*. A *goal* is simply an atom, and we use this terminology in the context of certain logic programming reasoning tasks. Since one can regard a clause $H \leftarrow B_1, \ldots, B_k$ as the logic formula $B_1 \wedge \cdots \wedge B_k \to H$, we say that a goal G is *derivable* in $\mathbb{L}$ if there exists a derivation of G with empty assumption using the clauses in $\mathbb{L}$.

The central task of logic programming is to check whether an atom G is *provable* in a program $\mathbb{L}$, in the sense that some substitution instance of G is derivable in $\mathbb{L}$. The key algorithm for this task is SLD-resolution, see e.g. [Llo87]. We use the notation $\mathbb{L} \vdash G$ to mean that G is provable in $\mathbb{L}$.

Probabilistic logic programming. We now recall the basics of PLP; the reader may consult [DRKT07b, DRKT07a] for a more comprehensive introduction. A probabilistic logic program $\mathbb{P}$ based on a logic program $\mathbb{L}$ assigns a probability label r to each clause $\mathcal{C}$ in $\mathbb{L}$, denoted as $\mathsf{Label}(\mathcal{C})$. One may also regard $\mathbb{P}$ as a set of probabilistic clauses of the form $r :: \mathcal{C}$, where $\mathcal{C}$ is a clause in $\mathbb{L}$, and each clause $\mathcal{C}$ is assigned a unique probability label r in $\mathbb{P}$. We also refer to $r :: \mathcal{C}$ simply as clauses.

Example 2.1. As our leading example we introduce the following probabilistic logic program $\mathbb{P}^{al}$. It models the scenario of Mary's house alarm, which is supposed to detect burglars, but it may be accidentally triggered by an earthquake. Mary may hear the alarm if she is awake, but even if the alarm is not sounding, in case she experiences an auditory hallucination (paracusia). The language of $\mathbb{P}^{al}$ includes 0-ary predicates Alarm, Eearthquake, Burglary, and 1-ary predicates Wake$(-)$, Hear_alarm$(-)$ and Paracusia$(-)$, and signature $\Sigma_{\mathsf{al}} = \{\mathsf{Mary}^0\}$ consisting of a constant. We do not have variables here, so $\mathbb{P}^{al}$ is a ground program. For readability we abbreviate Mary as M in the program.

0.01 ::	Earthquake	$\leftarrow$	0.01 ::	Paracusia(M)	$\leftarrow$
0.2 ::	Burglary	$\leftarrow$	0.6 ::	Wake(M)	$\leftarrow$
0.5 ::	Alarm	$\leftarrow$ Earthquake	0.8 ::	Hear_alarm(M)	$\leftarrow$ Alarm, Wake(M)
0.9 ::	Alarm	$\leftarrow$ Burglary	0.3 ::	Hear_alarm(M)	$\leftarrow$ Paracusia(M)

As a generalisation of the pure case, in probabilistic logic programming one is interested in the *probability* of a goal G being provable in a program $\mathbb{P}$. There are potentially multiple ways to define such probability — in this paper we focus on Sato's *distribution semantics* [Sat95] as below.

Given a probabilistic logic program $\mathbb{P} = \{p_1 :: \mathcal{C}_1, \ldots, p_n :: \mathcal{C}_n\}$, let $|\mathbb{P}|$ be its underlying pure logic program, namely $|\mathbb{P}| = \{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$. A *sub-program* $\mathbb{L}$ of $|\mathbb{P}|$ is a logic program consisting of a subset of the clauses in $|\mathbb{P}|$. This justifies using $\mathcal{P}(|\mathbb{P}|)$ to denote the set of all sub-programs of $|\mathbb{P}|$, and using $\mathbb{L} \subseteq |\mathbb{P}|$ to denote that $\mathbb{L}$ is a sub-program of $\mathbb{P}$. The central concept of the distribution semantics is that $\mathbb{P}$ determines a distribution $\mu_{\mathbb{P}}$ over the

sub-programs $\mathcal{P}(|\mathbb{P}|)$: for any $\mathbb{L} \in \mathcal{P}(|\mathbb{P}|)$,

$$\mu_{\mathbb{P}}(\mathbb{L}) \coloneqq \left(\prod_{\mathcal{C}_i \in \mathbb{L}} p_i\right) \cdot \left(\prod_{\mathcal{C}_j \in |\mathbb{P}| \setminus \mathbb{L}} (1 - p_j)\right)$$

where each p_i is the probability label of the clause $\mathcal{C}_i$ in $\mathbb{P}$. We simply refer to the value $\mu_{\mathbb{P}}(\mathbb{L})$ as the *probability* of the sub-program $\mathbb{L}$. For an arbitrary goal $G \in \mathsf{At}$, the *success probability* (or simply the probability) $\mathsf{Pr}_{\mathbb{P}}(G)$ of G w.r.t. program $\mathbb{P}$ is then defined as the sum of the probabilities of all the sub-programs of $\mathbb{P}$ in which G is provable:

$$\mathsf{Pr}_{\mathbb{P}}(G) \coloneqq \sum_{|\mathbb{P}| \supseteq \mathbb{L} \vdash G} \mu_{\mathbb{P}}(\mathbb{L}) = \sum_{|\mathbb{P}| \supseteq \mathbb{L} \vdash G} \left(\prod_{\mathcal{C}_i \in \mathbb{L}} p_i \cdot \prod_{\mathcal{C}_j \in |\mathbb{P}| \setminus \mathbb{L}} (1 - p_j)\right) \tag{2.1}$$

Intuitively one can regard every clause in $\mathbb{P}$ as a random event, then every sub-program $\mathbb{L}$ can be seen as a possible world over these events, and $\mu_{\mathbb{P}}$ is a distribution over all the possible worlds. The success probability $\mathsf{Pr}_{\mathbb{P}}(G)$ is then simply the probability of all the possible worlds in which the goal G is provable.

Example 2.2. For the program $\mathbb{P}^{al}$, consider the goal $\mathsf{Hear_alarm}(\mathsf{M})$. By the definition of distribution semantics (2.1), we can compute that the success probability $\mathsf{Pr}_{\mathbb{P}^{al}}(\mathsf{Hear_alarm}(\mathsf{M}))$ is ≈ 0.0911.

3. Ground case

In this section we introduce a coalgebraic semantics for *ground* probabilistic logic programming, i.e. for those programs where no variable appears. Our approach consists of two parts. First, we represent PLP programs as coalgebras (Subsection 3.1) and their executions as a final coalgebra semantics (Subsection 3.2) — this is a straightforward generalisation of the coalgebraic treatment of pure logic programs given in [KMP10]. Next, in Subsection 3.3 we investigate how to represent the distribution semantics as a final coalgebra, via a transformation of the coalgebra type of PLP programs. Appendix A shows how the success probability of a goal is effectively computable from the above representation.

3.1. **Coalgebraic Representation of PLP.**

A ground PLP program can be represented as a coalgebra for the composite $\mathcal{M}_{pr}\mathcal{P}_f \colon \mathbf{Sets} \to \mathbf{Sets}$ of the finite probability functor $\mathcal{M}_{pr} \colon \mathbf{Sets} \to \mathbf{Sets}$ and the finite powerset functor $\mathcal{P}_f \colon \mathbf{Sets} \to \mathbf{Sets}$. The definition of $\mathcal{M}_{pr}$ deserves some further explanation. It can be seen as the finite multiset functor based on the commutative monoid $([0,1], 0, \vee)$, where $\vee$ is the probabilistic 'or' for independent events defined as $a \vee b \coloneqq 1 - (1-a)(1-b)$. That is to say, on objects, $\mathcal{M}_{pr}(A)$ is the set of all *finite probability assignments* $\varphi \colon A \to [0,1]$, namely those φ with a finite support $\mathsf{supp}(\varphi) \coloneqq \{a \in A \mid \varphi(a) \neq 0\}$. For φ with support $\{a_1, \ldots, a_k\}$ and values $\varphi(a_i) = r_i$, it will often be convenient to use the standard notation $\varphi = r_1 a_1 + \cdots + r_k a_k$ or $\varphi = \sum_{i=1}^{k} r_i a_i$, where the purely formal '+' and '$\sum$' here should not be confused with the arithmetic addition. On morphisms, $\mathcal{M}_{pr}(h \colon A \to B)$ maps $\sum_{i=1}^{k} r_i a_i$ to $\sum_{i=1}^{k} r_i h(a_i)$.

Fix a ground probabilistic logic program $\mathbb{P}$ on a set of ground atoms At. The definition of $\mathbb{P}$ can be encoded as an $\mathcal{M}_{pr}\mathcal{P}_f$-coalgebra $p\colon \mathsf{At} \to \mathcal{M}_{pr}\mathcal{P}_f(\mathsf{At})$, as follows. Given $A \in \mathsf{At}$,

$$
\begin{array}{rrcl}
p(A)\colon & \mathcal{P}_f(\mathsf{At}) & \to & [0,1] \\
& \{B_1,\dots,B_n\} & \mapsto & \begin{cases} r & \text{if } r :: A \leftarrow B_1,\dots,B_n \text{ is a clause in } \mathbb{P} \\ 0 & \text{otherwise.} \end{cases}
\end{array}
$$

Or, equivalently, $p(A) \coloneqq \sum_{(r::A\leftarrow B_1,\dots,B_n)\ \in\ \mathbb{P}} r\{B_1,\dots,B_n\}$. Note that each $p(A)$ has a finite support because the program $\mathbb{P}$ consists of finitely many clauses.

Example 3.1. Consider program $\mathbb{P}^{al}$ from Example 2.1. The set of ground atoms At_{al} is $\{\mathsf{Alarm}, \mathsf{Earthquake}, \mathsf{Burgary}, \mathsf{Wake(M)}, \mathsf{Paracusia(M)}, \mathsf{Hear_alarm(M)}\}$. Here are some values of the corresponding coalgebra $p_{al}\colon \mathsf{At}_{al} \to \mathcal{M}_{pr}\mathcal{P}_f\mathsf{At}_{al}$:

$$
\begin{aligned}
p_{al}(\mathsf{Hear_alarm(M)}) &= 0.8\{\mathsf{Alarm}, \mathsf{Wake(M)}\} + 0.3\{\mathsf{Paracusia(M)}\} \\
p_{al}(\mathsf{Earthquake}) &= 0.01\{\}
\end{aligned}
$$

Remark 3.2. One might wonder why not simply adopt $\mathcal{P}_f(\mathcal{P}_f(-) \times [0,1])$ as the coalgebra type for PLP. The reason is that, although every ground PLP program generates a $\mathcal{P}_f(\mathcal{P}_f(-) \times [0,1])$-coalgebra, such encoding fails to be a 1-1 correspondence: a clause $\mathcal{C} \in \mathcal{P}_f(\mathsf{At})$ may be associated with different values in $[0,1]$, which violates the standard definition of PLP programs.

3.2. **Derivation Semantics.** In this section we are going to construct the final $\mathsf{At} \times \mathcal{M}_{pr}\mathcal{P}_f(-)$-coalgebra, thus providing a semantic interpretation for probabilistic logic programs based on At.

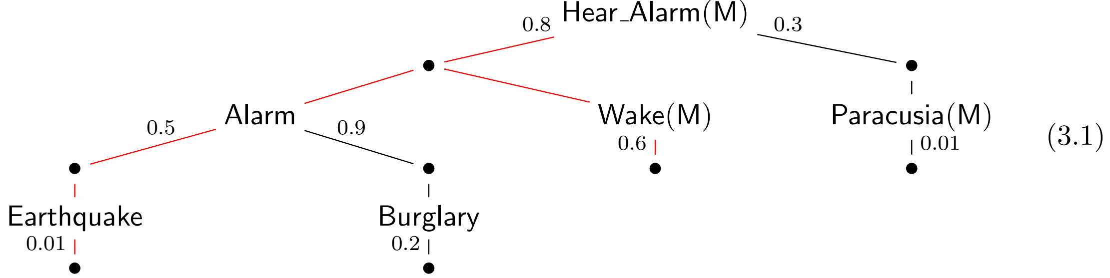
Before the technical developments, we give an intuitive view on the semantics that the final coalgebra is going to provide. We shall represent each goal as a *stochastic derivation tree* in the final $\mathsf{At} \times \mathcal{M}_{pr}\mathcal{P}_f(-)$-coalgebra. These trees are the probabilistic version of and-or derivation trees, which represent parallel SLD-resolutions for pure logic programming [GC94]. One can view a stochastic derivation trees as the unfolding of a goal under a PLP program.

Definition 3.3 (Stochastic derivation trees)**.** Given a ground PLP program $\mathbb{P}$ based on At, and an atom $A \in \mathsf{At}$, the *stochastic derivation tree* for A in $\mathbb{P}$ is the possibly infinite tree $\mathcal{T}$ such that:

(1) Every node is either an atom-node (labelled with an atom $A' \in \mathsf{At}$) or a clause-node (labelled with $\bullet$). These two types of nodes appear alternatingly in depth, in this order. In particular, the root is an atom-node labelled with A.
(2) Each edge from an atom-node to its (clause-)children is labelled with a probability value.
(3) Suppose s is an atom-node with label A'. Then for every clause $r :: A' \leftarrow B_1,\dots,B_k$ in $\mathbb{P}$, s has exactly one child t such that the edge $s \to t$ is labelled with r, and t has exactly k children labelled with $B_1,\dots,B_k$, respectively.

The final coalgebra semantics $[\![-]\!]_p$ for a program $\mathbb{P}$ will map a goal A to the stochastic derivation tree representing all possible SLD-resolutions of A in $\mathbb{P}$.

Example 3.4. Continuing Example 2.1, $[\![\mathsf{Hear_alarm(M)}]\!]_{p_{al}}$ is the stochastic derivation tree below. The subtree highlighted in red represents one of the successful proofs of $\mathsf{Hear_alarm(M)}$ in p_{al}: indeed, note that a single child is selected for each atom-node A (corresponding to a clause matching A), all children of any clause-node are selected (corresponding to the atoms in the body of the clause), and the subtree has clause-nodes as leaves (all atoms are proven).



(3.1)

Any such subtree describes a proof, but does not yield a probability value to be associated to a goal — this is the remit of the distribution semantics, see Example 3.9 below.

In the remaining part of the section, we construct the cofree coalgebra for $\mathcal{M}_{pr}\mathcal{P}_f$ via a so-called terminal sequence [Wor99], and obtain $[\![-]\!]_p$ from the resulting universal property. We report the steps of the terminal sequence as they are instrumental in showing that the elements of the cofree coalgebra can be seen as stochastic derivation trees.

Construction 3.5. The terminal sequence for the functor $\mathsf{At} \times \mathcal{M}_{pr}\mathcal{P}_f(-) : \mathbf{Sets} \to \mathbf{Sets}$ consists of sequences of objects $\{X_\alpha\}_{\alpha \in \mathbf{Ord}}$ and arrows $\{\delta^\alpha_\beta \colon X_\alpha \to X_\beta\}_{\beta < \alpha \in \mathbf{Ord}}$ constructed by the following transfinite induction:

$$X_\alpha := \begin{cases} \mathsf{At} & \alpha = 0 \\ \mathsf{At} \times \mathcal{M}_{pr}\mathcal{P}_f(X_\xi) & \alpha = \xi + 1 \\ \lim\{\delta^\chi_\xi \mid \xi < \chi < \alpha\} & \alpha \text{ is limit} \end{cases}$$

$$\delta^\alpha_\beta := \begin{cases} \pi_1 & \alpha = 1, \beta = 0 \\ id_{\mathsf{At}} \times \mathcal{M}_{pr}\mathcal{P}_f(\delta^{\xi+1}_\xi) & \alpha = \beta + 1 = \xi + 2 \\ \text{the limit projections} & \alpha \text{ is limit}, \beta < \alpha \\ \text{universal map to } X_\beta & \beta \text{ is limit}, \alpha = \beta + 1 \end{cases}$$

Proposition 3.6. *The terminal sequence for the functor* $\mathsf{At} \times \mathcal{M}_{pr}\mathcal{P}_f(-)$ *converges to a limit* X_γ *such that* $X_\gamma \cong X_{\gamma+1}$.

Proof. We apply the following result from [Wor99]:

Proposition 3.7 [Wor99, Corollary 3.3]**.** *If T is an accessible endofunctor on a locally presentable category, and if T preserves monics, then the terminal T-sequence $\{A_\alpha, f^\alpha_\beta\}$ converges, necessarily to a terminal T-coalgebra.*

Since **Sets** is a locally presentable category, it remains to verify the two conditions for $\mathcal{M}_{pr}\mathcal{P}_f$. It is well-known that $\mathcal{P}_f$ is ω-accessible, and $\mathcal{M}_{pr}$ has the same property, see e.g. [Sil10, Prop. 6.1.2]. Because accessibility is defined in terms of colimit preservation, it is clearly preserved by composition, and thus $\mathcal{M}_{pr}\mathcal{P}_f$ is also accessible. It remains to check that it preserves monics. For $\mathcal{M}_{pr}$, given any monomorphism $i : C \to D$ in **Sets**, suppose $\mathcal{M}_{pr}(i)(\varphi) = \mathcal{M}_{pr}(i)(\varphi')$ for some $\varphi, \varphi' \in \mathcal{M}_{pr}(C)$. Then for any $d \in D$,

$\mathcal{M}_{pr}(i)(\varphi)(d) = \mathcal{M}_{pr}(i)(\varphi')(d)$. If we focus on the image $i[C]$, then there is an inverse function $i^{-1} : i[C] \to C$, and $\mathcal{M}_{pr}(i)(\varphi) = \mathcal{M}_{pr}(i)(\varphi')$ implies that $\varphi(i^{-1}(d)) = \varphi'(i^{-1}(d))$ for any $d \in i[C]$. But this simply means that $\varphi = \varphi'$. As the same is true for $\mathcal{P}_f$ and the property is preserved by composition, we have that $\mathcal{M}_{pr}\mathcal{P}_f$ preserves monics. Therefore we can conclude that the terminal sequence for $At \times \mathcal{M}_{pr}\mathcal{P}_f$ converges to the cofree $\mathcal{M}_{pr}\mathcal{P}_f$-coalgebra on At. □

Note that $X_{\gamma+1}$ is defined as $\mathsf{At} \times \mathcal{M}_{pr}\mathcal{P}_f(X_\gamma)$, and the above isomorphism makes $X_\gamma \to \mathsf{At} \times \mathcal{M}_{pr}\mathcal{P}_f(X_\gamma)$ the final $\mathsf{At} \times \mathcal{M}_{pr}\mathcal{P}_f$-coalgebra — or, in other words, *cofree $\mathcal{M}_{pr}\mathcal{P}_f$-coalgebra* on At. As for the tree representation of the elements of X_γ, recall that elements of the cofree $\mathcal{P}_f\mathcal{P}_f$-coalgebra on At can be seen as and-or trees [KMP10]. By replacing the first $\mathcal{P}_f$ with $\mathcal{M}_{pr}$, effectively one adds probability values to the edges from and-nodes to or-nodes (which are edges from atom-nodes to clause-nodes in our stochastic derivation trees), as in (3.1). Thus stochastic derivation trees as in Definition 3.3 are elements of X_γ. The action of the coalgebra map $\cong: X_\gamma \to \mathsf{At} \times \mathcal{M}_{pr}\mathcal{P}_f(X_\gamma)$ is best seen with an example: the tree $\mathcal{T}$ in (3.1) (as an element of X_γ) is mapped to the pair $\langle \mathsf{Hear_alarm(M)}, \varphi\rangle$, where φ is the function $\mathcal{P}_f(X_\gamma) \to [0,1]$ assigning 0.8 to the set consisting of the subtrees of $\mathcal{T}$ with root Alarm and with root $\mathsf{Wake(M)}$, 0.3 to the singleton consisting to the subtree of $\mathcal{T}$ with root $\mathsf{Paracusia(M)}$, and 0 to any other finite set of trees.

With all the definitions at hand, it is straightforward to check that $[\![-]\!]_p$ mapping $A \in \mathsf{At}$ to its stochastic derivation tree in $\mathbb{P}$ makes the following diagram commute

$$\begin{array}{ccc} \mathsf{At} & \overset{[\![-]\!]_p}{\dashrightarrow} & X_\gamma \\ \downarrow{\scriptstyle <id,p>} & & \downarrow{\scriptstyle \cong} \\ \mathsf{At} \times \mathcal{M}_{pr}\mathcal{P}_f(\mathsf{At}) & \xrightarrow{id \times \mathcal{M}_{pr}\mathcal{P}_f([\![-]\!]_p)} & \mathsf{At} \times \mathcal{M}_{pr}\mathcal{P}_f(X_\gamma) \end{array}$$

and thus by uniqueness it coincides with the $\mathsf{At} \times \mathcal{M}_{pr}\mathcal{P}_f$-coalgebra map provided by the universal property of the final $\mathsf{At} \times \mathcal{M}_{pr}\mathcal{P}_f$-coalgebra $X_\gamma \to \mathsf{At} \times \mathcal{M}_{pr}\mathcal{P}_f(X_\gamma)$.

3.3. Distribution Semantics.

This section gives a coalgebraic representation of the usual *distribution semantics* of probabilistic logic programming. As in the previous section, before the technical developments we gather some preliminary intuition. Recall from Section 2 that the core of the distribution semantics is the probability distribution over the sub-programs (subsets of clauses) of a given program $\mathbb{P}$. These sub-programs are also called (possible) worlds, and the distribution semantics of a goal is the sum of the probabilities of all the worlds in which it is provable.

In order to encode this information as elements of a final coalgebra, we need to present it in tree-shape. Roughly speaking, we form a distribution over the sub-programs along the execution tree. This justifies the following notion of *distribution trees*.

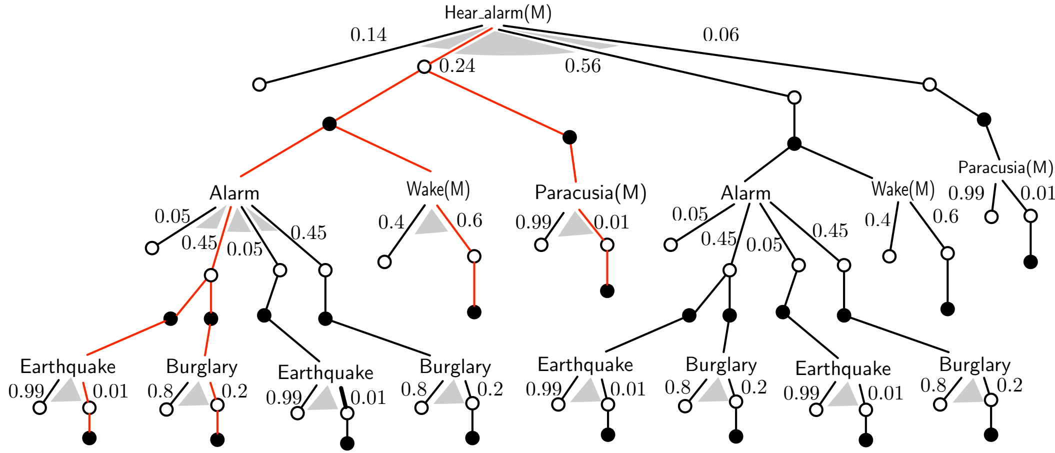
Definition 3.8 (Distribution trees). Given a PLP program $\mathbb{P}$ and an atom A, the *distribution tree* for A in $\mathbb{P}$ is the possibly infinite tree $\mathcal{T}$ satisfying the following properties:

(1) Every node is exactly one of the three kinds: atom-node (labelled with an atom $A' \in \mathsf{At}$), world-node (labelled with $\circ$), clause-node (labelled with $\bullet$). They appear alternatingly in this order in depth. In particular the root is an atom-node labelled with A.
(2) Every edge from an atom-node to its (world-)children is labelled with a probability value, and the labels on all the edges (if exist) from the same atom-node sum up to 1.

(3) Suppose s is an atom-node labelled with A', and $C = \{\mathcal{C}_1, \ldots, \mathcal{C}_m\}$ is the set of all the clauses in $\mathbb{P}$ whose head is A'. Then s has 2^m (world-)children, each standing for a subset X of C. If a child t stands for X, then the edge $s \to t$ is labelled with probability $\prod_{\mathcal{C} \in X} \mathsf{Label}(\mathcal{C}) \cdot \prod_{\mathcal{C}' \in C \setminus X} (1 - \mathsf{Label}(\mathcal{C}'))$ — recall that $\mathsf{Label}(\mathcal{C})$ is the probability labelling $\mathcal{C}$. Also, t has exactly $|X|$-many (clause-)children, each standing for a clause $\mathcal{C} \in X$. If a child u stands for $\mathcal{C} = r :: A' \leftarrow B_1, \ldots, B_k$, then u has k (atom-)children, labelled with $B_1, \ldots, B_k$ respectively.

Comparing distribution trees with stochastic derivation trees (Definition 3.3), one can observe the addition of another class of nodes, representing possible worlds. Moreover, the possible worlds associated with an atom-node must form a probability distribution — as opposed to stochastic derivation trees, in which probabilities labelling parallel edges do not need to share any relationship. An example of the distribution tree associated with a goal is provided in the continuation of our leading example (Examples 2.1 and 3.4).

Example 3.9. In the context of Example 2.1, the distribution tree of $\mathsf{Hear_alarm(M)}$ is depicted below, where we use grey shades to emphasise sets of edges expressing a probability distribution. Also, note the $\circ$s with no children, standing for empty worlds (namely worlds containing no clause).



In the literature, the distribution semantics usually associates with a goal a single probability value (2.1), rather than a whole tree. However, given the distribution tree it is straightforward to compute such probability. The subtree highlighted in red above describes a refutation of $\mathsf{Hear_alarm(M)}$ with probability 0.000001296 (= the product of all the probabilities in the subtree). The sum of all the probabilities associated to such 'proof' subtrees yields the usual distribution semantics (2.1) — the computation is shown in detail in Appendix A.

In the remainder of this section, we focus on the coalgebraic characterisation of distribution trees and the associated semantics map. Our strategy will be to introduce a novel coalgebra type $\mathcal{D}_{\leq 1}\mathcal{P}_f\mathcal{P}_f$, such that distribution trees can be seen as elements of the cofree coalgebra. Then, we will provide a natural transformation $\mathcal{M}_{pr} \Rightarrow \mathcal{D}_{\leq 1}\mathcal{P}_f$, which can be used to transform stochastic derivation trees into distribution trees. Finally, composing the universal properties of these cofree coalgebras will yield the desired distribution semantics.

We begin with the definition of $\mathcal{D}_{\leq 1}\mathcal{P}_f$. This is simply the composite $\mathcal{D}_{\leq 1}\mathcal{P}_f \colon \mathbf{Sets} \to \mathbf{Sets}$, where $\mathcal{D}_{\leq 1}$ is the *sub-probability distribution* functor. Recall that $\mathcal{D}_{\leq 1}$ maps X to the

set of sub-probability distributions with finite supports on X (i.e., convex combinations of elements of X whose sum is less or equal to 1), and acts component-wise on functions.

Remark 3.10. Note that we cannot work with full probabilities $\mathcal{D}$ here, since a goal may not match any clause. In such a case there is no world in which the goal is provable and its probability in the program is 0.

The next step is to recover distribution trees as the elements of the $\mathcal{D}_{\leq 1}\mathcal{P}_f\mathcal{P}_f$-cofree coalgebra on At. This goes via a terminal sequence, similarly to the case of $\mathcal{M}_{pr}\mathcal{P}_f$ in the previous section. We will not go into full detail, but only mention that the terminal sequence for $\mathsf{At} \times \mathcal{D}_{\leq 1}\mathcal{P}_f\mathcal{P}_f(-) : \mathbf{Sets} \to \mathbf{Sets}$ is constructed as the one for $\mathsf{At} \times \mathcal{M}_{pr}\mathcal{P}_f(-) : \mathbf{Sets} \to \mathbf{Sets}$ (Construction 3.5), with $\mathcal{D}_{\leq 1}\mathcal{P}_f$ replacing $\mathcal{M}_{pr}$.

Proposition 3.11. *The terminal sequence of* $\mathsf{At} \times \mathcal{D}_{\leq 1}\mathcal{P}_f\mathcal{P}_f(-)$ *converges at some limit ordinal* χ, *and* $(\lambda_\chi^{\chi+1})^{-1} : Y_\chi \to \mathsf{At} \times \mathcal{D}_{\leq 1}\mathcal{P}_f\mathcal{P}_f Y_\chi$ *is the final* $\mathsf{At} \times \mathcal{D}_{\leq 1}\mathcal{P}_f\mathcal{P}_f$ *coalgebra.*

Proof. As for Proposition 3.6, by [Wor99, Cor. 3.3] it suffices to show that $\mathcal{D}_{\leq 1}\mathcal{P}_f$ is accessible and preserves monos. Both are simple exercises; in particular, see [BSdV04] for accessibility of $\mathcal{D}_{\leq 1}$. □

The association of distribution trees with elements of Y_χ is suggested by the type $\mathsf{At} \times \mathcal{D}_{\leq 1}\mathcal{P}_f\mathcal{P}_f$. Indeed, $\mathsf{At} \times \mathcal{D}_{\leq 1}$ is the layer of atom-nodes, labelled with elements of At and with outgoing edges forming a sub-probability distribution; the first $\mathcal{P}_f$ is the layer of world-nodes; the second $\mathcal{P}_f$ is the layer of clause-nodes. The coalgebra map $Y_\chi \to \mathsf{At} \times \mathcal{D}_{\leq 1}\mathcal{P}_f\mathcal{P}_f Y_\chi$ associates a goal to subtrees of its distribution trees, analogously to the coalgebra structure on stochastic derivation trees in the previous section.

The last ingredient we need is a translation of stochastic derivation trees into distribution trees. We formalise this as a natural transformation $\mathsf{pw} : \mathcal{M}_{pr} \Rightarrow \mathcal{D}_{\leq 1}\mathcal{P}_f$. The naturality of pw can be checked with a simple calculation.

Definition 3.12. The '**p**ossible **w**orlds' natural transformation $\mathsf{pw}\colon \mathcal{M}_{pr} \Rightarrow \mathcal{D}_{\leq 1}\mathcal{P}_f$ is defined by $\mathsf{pw}_X\colon \varphi \mapsto \sum_{Y \subseteq \mathsf{supp}(\varphi)} r_Y Y$, where each $r_Y = \prod_{y \in Y} \varphi(y) \cdot \prod_{y' \in \mathsf{supp}(\varphi)\setminus Y}(1 - \varphi(y'))$. In particular, when $\mathsf{supp}(\varphi)$ is empty, $\mathsf{pw}_X(\varphi)$ is the empty sub-distribution $\varnothing$.

Now we have all the ingredients to characterise the distribution semantics coalgebraically, as the morphism $\langle\!\langle - \rangle\!\rangle_p \colon \mathsf{At} \to Y_\chi$ defined by the following diagram, which maps $A \in \mathsf{At}$ to its distribution tree in p.

$$
\begin{array}{ccccc}
\mathsf{At} & \overset{[\![-]\!]_p}{\dashrightarrow} & X_\gamma & \overset{!}{\dashrightarrow} & Y_\chi \\
\downarrow{\scriptstyle <id_{\mathsf{At}},p>} & & \downarrow{\scriptstyle \cong} & & \\
\mathsf{At} \times \mathcal{M}_{pr}\mathcal{P}_f\mathsf{At} & \xrightarrow{id_{\mathsf{At}} \times \mathcal{M}_{pr}\mathcal{P}_f([\![-]\!]_p)} & \mathsf{At} \times \mathcal{M}_{pr}\mathcal{P}_f X_\gamma & & \downarrow{\scriptstyle \cong} \\
\downarrow{\scriptstyle id_{\mathsf{At}} \times \mathsf{pw}_{\mathcal{P}_f(\mathsf{At})}} & & \downarrow{\scriptstyle id_{\mathsf{At}} \times \mathsf{pw}_{\mathcal{P}_f(X_\gamma)}} & & \\
\mathsf{At} \times \mathcal{D}_{\leq 1}\mathcal{P}_f\mathcal{P}_f\mathsf{At} & \xrightarrow{id_{\mathsf{At}} \times \mathcal{D}_{\leq 1}\mathcal{P}_f\mathcal{P}_f([\![-]\!]_p)} & \mathsf{At} \times \mathcal{D}_{\leq 1}\mathcal{P}_f\mathcal{P}_f X_\gamma & \xrightarrow{id_{\mathsf{At}} \times \mathcal{D}_{\leq 1}\mathcal{P}_f\mathcal{P}_f(!)} & \mathsf{At} \times \mathcal{D}_{\leq 1}\mathcal{P}_f\mathcal{P}_f Y_\chi
\end{array}
\tag{3.2}
$$

(with the dashed arrow $\langle\!\langle - \rangle\!\rangle_p \colon \mathsf{At} \dashrightarrow Y_\chi$ drawn over the top row)

Note the use of pw to extend probabilistic logic programs and stochastic derivation trees to the same coalgebra type as distribution trees. Then the distribution semantics $\langle\!\langle - \rangle\!\rangle_p$ is uniquely defined by the universal property of the final $\mathsf{At} \times \mathcal{D}_{\leq 1}\mathcal{P}_f\mathcal{P}_f$-coalgebra. By uniqueness, it can also be computed as the composite $! \circ [\![-]\!]_p$, that is, first one derives the semantics $[\![-]\!]_p$, then applies the translation pw to each level of the resulting stochastic derivation tree, in order to turn it into a distribution tree.

4. General Case

We now generalise our coalgebraic treatment to arbitrary probabilistic logic programs and goals, possibly including variables. The section has a similar structure as the one devoted to the ground case. First, in Subsection 4.1, we give a coalgebraic representation for general PLP, and equip it with a final coalgebra semantics in terms of stochastic derivation trees (Subsection 4.2). Next, in Subsection 4.3, we study the coalgebraic representation of the distribution semantics. We begin by introducing our leading example — an extension of Example 2.1.

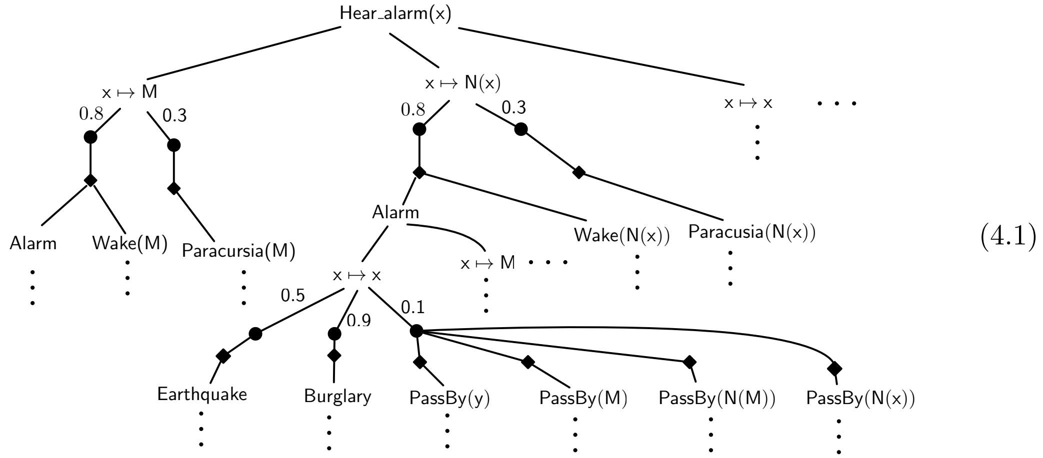
Example 4.1. We tweak the ground program of Example 2.1. Now it is not just Mary that may hear the alarm, but also her neighbours. There is a small probability that the alarm rings because someone passes too close to Mary's house. However, we can only estimate the possibility of paracusia and being awake for Mary, not the neighbours. The revised program, which by abuse of notation we also call $\mathbb{P}^{al}$, is based on an extension of the language in Example 2.1: we add a new 1-ary function symbol Neigh^1 to the signature Σ_{al}, and a new 1-ary predicate $\mathsf{PassBy}(-)$ to the alphabet. Note the appearance of a variable x.

$$
\begin{array}{rll|rll}
0.01 :: & \mathsf{Earthquake} & \leftarrow & 0.5 :: & \mathsf{Alarm} & \leftarrow \mathsf{Earthquake} \\
0.2 :: & \mathsf{Burglary} & \leftarrow & 0.9 :: & \mathsf{Alarm} & \leftarrow \mathsf{Burglary} \\
0.6 :: & \mathsf{Wake(Mary)} & \leftarrow & 0.1 :: & \mathsf{Alarm} & \leftarrow \mathsf{PassBy(x)} \\
0.01 :: & \mathsf{Paracusia(Mary)} & \leftarrow & 0.3 :: & \mathsf{Hear_alarm(x)} & \leftarrow \mathsf{Paracusia(x)} \\
0.8 :: & \mathsf{Wake(Neigh(x))} & \leftarrow \mathsf{Wake(x)} & 0.8 :: & \mathsf{Hear_alarm(x)} & \leftarrow \mathsf{Alarm},\ \mathsf{Wake(x)}
\end{array}
$$

As we want to maintain our approach a direct generalisation of the coalgebraic semantics [BZ15] for pure logic programs, the derivation semantics $[\![-]\!]$ for general PLP will represent resolution by *unification*. This means that, at each step of the computation, given a goal A, one seeks substitutions θ, τ such that $A\theta = H\tau$ for the head H of some clause in the program. As a roadmap, we anticipate the way this computation is represented in terms of stochastic derivation trees (Definition 4.7 below), with a continuation of our leading example.

Example 4.2. In the context of Example 4.1, the tree for $[\![\mathsf{Hear_alarm(x)}]\!]_{\mathbb{P}^{al}}$ is (partially) depicted below. Compared to the ground case (Example 3.4), now substitutions applied on the goal side appear explicitly as labels. Moreover, note that for each atom $A\theta$ and clause $\mathcal{C}$, there might be multiple substitutions τ such that $A\theta = \mathsf{Head}(\mathcal{C})\tau$. This is presented in the following tree (4.1) by the ⧫-labelled nodes, which are children of the clause-nodes (labelled

with •). We abbreviate Neigh as N and Mary as M.



(4.1)

Resolution by unification as above will be implemented in two stages. The first step is devising a map for term-matching. Assuming that the substitution instance $A\theta$ of a goal A is already given, we define p performing term-matching of $A\theta$ in a given program $\mathbb{P}$:

$$p(A\theta)\colon \quad \{B_1\tau_i, \dots, B_k\tau_i\}_{i\in I\subseteq\mathbb{N}} \quad \mapsto \begin{cases} r & (r :: H \leftarrow B_1, \dots, B_k) \in \mathbb{P} \text{ and} \\ & \quad I \text{ contains all } i \text{ s.t. } A\theta = H\tau_i \\ 0 & \text{otherwise.} \end{cases} \tag{4.2}$$

Intuitively, one application of such map is represented in a tree structure as Example 4.2 by the first two layers of the subtree rooted at θ. The reason why the domain of $p(A)$ is a *countable* set $\{B_1\tau_i, \dots, B_k\tau_i\}_{i\in I\subseteq\mathbb{N}}$ of instances of the same body $B_1, \dots, B_k$ is that the same clause may match a goal with countably many different substitutions τ_i. For example in the bottom part of (4.1) there are countably infinite substitutions τ_i matching the head of $\mathsf{Alarm} \leftarrow \mathsf{PassBy(x)}$ to the goal Alarm, substituting x with $\mathsf{Mary}, \mathsf{Neigh(Mary)}, \mathsf{Neigh(x)}, \dots$. This will be reflected in the coalgebraic representation of PLP (see (4.4) below) by the use of the countable powerset functor $\mathcal{P}_c$.

In order to model arbitrary unification, the second step is considering all substitutions θ on the goal A such that a term-matcher for $A\theta$ exists. There is an elegant categorical construction [BZ15] packing together these two steps into a single coalgebra map. We will start with this in the next session.

4.1. **Coalgebraic Representation of PLP.** Towards a categorification of general PLP, the first concern is to account for the presence of variables in atoms. This is standardly done by letting the space of atoms on an alphabet $\mathcal{A}$ be a presheaf $\mathsf{At}\colon \mathbf{L}_\Sigma^{\mathrm{op}} \to \mathbf{Sets}$ rather than a set. Here the index category $\mathbf{L}_\Sigma^{\mathrm{op}}$ is the opposite *Lawvere Theory* of Σ (see Section 2). For each $n \in \mathsf{Ob}(\mathbf{L}_\Sigma^{\mathrm{op}})$, $\mathsf{At}(n)$ is defined as the set of $\mathcal{A}$-atoms in context n. Given an n-tuple $\theta = \langle t_1, \dots, t_n\rangle \in \mathbf{L}_\Sigma^{\mathrm{op}}[n, m]$ of Σ-terms in context m, $\mathsf{At}(\theta)\colon \mathsf{At}(n) \to \mathsf{At}(m)$ is defined by substitution, namely $\mathsf{At}(\theta)(A) = A\theta$, for any $A \in \mathsf{At}(n)$.

As observed in [KP11] for pure logic programs, if we naively try to model our specification (4.2) for p as a coalgebra on At, we run into problems: indeed p is not a natural transformation, thus not a morphism between presheaves. Intuitively, this is because the existence of a term-matching for a substitution instance $A\theta$ of A does not necessarily imply the existence

of a term-matching for A itself. For pure logic programs, this problem can be solved in at least two ways. First, [KP11] relaxes naturality by changing the base category of presheaves from **Sets** to **Poset**. We take here the second route, namely give a 'saturated' coalgebraic treatment of PLP, generalising the modelling of pure logic programs proposed in [BZ15]. This approach has the advantage of letting us work with **Sets**-based presheaves, and be still able to recover term-matching via a 'desaturation' operation — see [BZ15] and Appendix B.

The Saturation Adjunction. To this aim, we briefly recall the saturated approach from [BZ15]. The central piece is the adjunction $\mathcal{U} \dashv \mathcal{K}$ on presheaf categories, as on the left below.

$$\mathbf{Sets}^{\mathbf{L}_\Sigma^{\mathrm{op}}} \underset{\mathcal{K}}{\overset{\mathcal{U}}{\rightleftarrows}} \mathbf{Sets}^{|\mathbf{L}_\Sigma^{\mathrm{op}}|} \qquad\qquad \begin{array}{ccc} |\mathbf{L}_\Sigma^{\mathrm{op}}| & \overset{\iota}{\hookrightarrow} & \mathbf{L}_\Sigma^{\mathrm{op}} \\ \mathsf{F}\downarrow & \swarrow \mathcal{K}(\mathsf{F}) & \\ \mathbf{Sets} & & \end{array} \tag{4.3}$$

Here $|\mathbf{L}_\Sigma^{\mathrm{op}}|$ is the discretisation of $\mathbf{L}_\Sigma^{\mathrm{op}}$, i.e. all the arrows but the identities are dropped. The left adjoint $\mathcal{U}$ is the forgetful functor, given by precomposition with the obvious inclusion $\iota\colon |\mathbf{L}_\Sigma^{\mathrm{op}}| \to \mathbf{L}_\Sigma^{\mathrm{op}}$. $\mathcal{U}$ has a right adjoint $\mathcal{K} = \mathsf{Ran}\iota\colon \mathbf{Sets}^{|\mathbf{L}_\Sigma^{\mathrm{op}}|} \to \mathbf{Sets}^{\mathbf{L}_\Sigma^{\mathrm{op}}}$, which sends every presheaf $\mathsf{F}\colon |\mathbf{L}_\Sigma^{\mathrm{op}}| \to \mathbf{Sets}$ to its *right Kan extension* along ι, as in the rightmost diagram in (4.3). The definition of $\mathcal{K}$ can be computed [Mac71] as follows:

- On objects $\mathsf{F} \in \mathsf{Ob}(\mathbf{Sets}^{\mathbf{L}_\Sigma^{\mathrm{op}}})$, the presheaf $\mathcal{K}(\mathsf{F})\colon \mathbf{L}_\Sigma^{\mathrm{op}} \to \mathbf{Sets}$ is defined by letting $\mathcal{K}(\mathsf{F})(n)$ be the product $\prod_{\theta \in \mathbf{L}_\Sigma^{\mathrm{op}}[n,m]} \mathsf{F}(m)$, where m ranges over $\mathsf{Ob}(\mathbf{L}_\Sigma^{\mathrm{op}})$. Intuitively, every element in $\mathcal{K}(\mathsf{F})(n)$ is a tuple with index set $\bigcup_{m \in \mathsf{Ob}(\mathbf{L}_\Sigma^{\mathrm{op}})} \mathbf{L}_\Sigma^{\mathrm{op}}[n,m]$, and its component at index $\theta\colon n \to m$ must be an element in $\mathsf{F}(m)$. We follow the convention of [BZ15] and write $\dot{x}, \dot{y}, \dots$ for such tuples, and $\dot{x}(\theta)$ for the component of $\dot{x}$ at index θ.
  With this convention, given an arrow $\sigma \in \mathbf{L}_\Sigma^{\mathrm{op}}[n,n']$, $\mathcal{K}(\mathsf{F})(\sigma)$ is defined by pointwise substitution as the mapping of the tuple $\dot{x}$ to the tuple $\langle \dot{x}(\theta \circ \sigma)\rangle_{\theta\colon n' \to m}$.
- On arrows, given a morphism $\alpha\colon \mathsf{F} \to \mathsf{G}$ in $\mathbf{Sets}^{|\mathbf{L}_\Sigma^{\mathrm{op}}|}$, $\mathcal{K}(\alpha)$ is a natural transformation $\mathcal{K}(\mathsf{F}) \to \mathcal{K}(\mathsf{G})$ defined pointwisely as $\mathcal{K}(\alpha)(n)\colon \dot{x} \mapsto \langle \alpha_m(\dot{x}(\theta))\rangle_{\theta\colon n\to m}$.

It is also useful to record the unit $\eta\colon 1 \to \mathcal{KU}$ of the adjunction $\mathcal{U} \dashv \mathcal{K}$. Given a presheaf $\mathsf{F}\colon \mathbf{L}_\Sigma^{\mathrm{op}} \to \mathbf{Sets}$, $\eta_{\mathsf{F}}\colon \mathsf{F} \to \mathcal{KU}\mathsf{F}$ is a natural transformation defined by $\eta_{\mathsf{F}}(n)\colon x \mapsto \langle \mathsf{F}(\theta)(x)\rangle_{\theta\colon n\to m}$.

Saturation in PLP. We now construct the coalgebra structure on the presheaf At modelling PLP. First, we are now able to represent p in (4.2) as a coalgebra map. The aforementioned naturality issue is solved by defining it as a morphism in $\mathbf{Sets}^{|\mathbf{L}_\Sigma^{\mathrm{op}}|}$ rather than in $\mathbf{Sets}^{\mathbf{L}_\Sigma^{\mathrm{op}}}$, thus making naturality trivial. The coalgebra p will have the following type

$$p\colon \mathcal{U}\mathsf{At} \to \widehat{\mathcal{M}_{pr}}\widehat{\mathcal{P}_c}\widehat{\mathcal{P}_f}\mathcal{U}\mathsf{At} \tag{4.4}$$

where $\widehat{(\cdot)}$ is the obvious extension of **Sets**-endofunctors to $\mathbf{Sets}^{|\mathbf{L}_\Sigma^{\mathrm{op}}|}$-endofunctors, defined by functor precomposition. With respect to the ground case, note the insertion of $\widehat{\mathcal{P}_c}$, the lifting of the *countable* powerset functor, in order to account for the countably many instances of a clause that may match the given goal (*cf.* the discussion below (4.2)).

Example 4.3. Our program $\mathbb{P}^{al}$ (Example 4.1) is based on $\mathsf{At}_{al}\colon \mathbf{L}_{\Sigma_{\mathsf{al}}}^{\mathrm{op}} \to \mathbf{Sets}$. Some of its values are $\mathsf{At}_{al}(0) = \{\mathsf{Mary}, \mathsf{Neigh}(\mathsf{Mary}), \mathsf{Neigh}(\mathsf{Neigh}(\mathsf{Mary})), \dots\}$ and $\mathsf{At}_{al}(1) = \{\mathsf{x}, \mathsf{Mary}, \mathsf{Neigh}(\mathsf{x}), \mathsf{Neigh}(\mathsf{Mary}), \dots\}$. Part of the coalgebra p_{al} modelling the program $\mathbb{P}^{al}$ is as follows (*cf.* the tree (4.1)). For space reason, we abbreviate Mary as M in the second

equation.

$$\begin{aligned}(p_{al})_0(\mathsf{Hear_alarm}(\mathsf{Mary})) &= 0.8\{\{\mathsf{Alarm}, \mathsf{Wake}(\mathsf{Mary})\}\} + 0.3\{\{\mathsf{Parasusia}(\mathsf{Mary})\}\} \\ (p_{al})_1(\mathsf{Alarm}) &= 0.5\{\{\mathsf{Earthquake}\}\} + 0.9\{\{\mathsf{Burglary}\}\} \\ &\quad + 0.1\{\{\mathsf{PassBy}(\mathsf{M})\}, \{\mathsf{PassBy}(\mathsf{Neigh}(\mathsf{M}))\}, \{\mathsf{PassBy}(\mathsf{Neigh}(\mathsf{x}))\}, \dots\}\end{aligned}$$

The universal property of the adjunction (4.3) gives a canonical 'lifting' of p to a $\mathcal{K}\widehat{\mathcal{M}}_{pr}\widehat{\mathcal{P}_c}\widehat{\mathcal{P}_f}\mathcal{U}$-coalgebra $p^\sharp$ on At, performing unification rather than just term-matching:

$$p^\sharp := \mathsf{At} \xrightarrow{\eta_{\mathsf{At}}} \mathcal{K}\mathcal{U}\mathsf{At} \xrightarrow{\mathcal{K}p} \mathcal{K}\widehat{\mathcal{M}}_{pr}\widehat{\mathcal{P}_c}\widehat{\mathcal{P}_f}\mathcal{U}\mathsf{At} \tag{4.5}$$

where η is the unit of the adjunction, as defined above. Spelling it out, $p^\sharp$ is the mapping

$$p^\sharp_n \ : \ A \in \mathsf{At}(n) \mapsto \langle p_m(A\theta)\rangle_{\theta\colon n\to m}$$

Intuitively, $p^\sharp_n$ retrieves all the unifiers $\langle\theta, \tau\rangle$ of A and head H in $\mathbb{P}$: first, we have $A\theta \in \mathsf{At}(m)$ as a component of the saturation of A by η_{At}; then we term-match H with $A\theta$ by $\mathcal{K}p_m$.

Remark 4.4. Note that the parameter $n \in \mathsf{Ob}(\mathbf{L}_\Sigma^{\mathrm{op}})$ in the natural transformation $p^\sharp$ fixes the pool $\{x_1, \dots, x_n\}$ of variables appearing in the atoms (and relative substitutions) that are considered in the computation. Analogously to the case of pure logic programs [KP11, BZ15], it is intended that such n can always be chosen 'big enough' so that all the relevant substitution instances of the current goal and clauses in the program are covered — note the variables occurring therein always form a *finite* set, included in $\{x_1, \dots, x_m\}$ for some $m \in \mathbb{N}$.

4.2. **Derivation Semantics.** Once we have identified our coalgebra type, the construction leading to the derivation semantics $[\![-]\!]_{p^\sharp}$ for general PLP is completely analogous to the ground case. One can define the cofree coalgebra for $\mathcal{K}\widehat{\mathcal{M}}_{pr}\widehat{\mathcal{P}_c}\widehat{\mathcal{P}_f}\mathcal{U}(-)$ by terminal sequence, similarly to Construction 3.5. For simplicity, henceforth we denote the functor $\mathcal{K}\widehat{\mathcal{M}}_{pr}\widehat{\mathcal{P}_c}\widehat{\mathcal{P}_f}\mathcal{U}(-)$ by $\mathcal{S}$.

Construction 4.5. The terminal sequence for $\mathsf{At} \times \mathcal{S}(-) : \mathbf{Sets}^{\mathbf{L}_\Sigma^{\mathrm{op}}} \to \mathbf{Sets}^{\mathbf{L}_\Sigma^{\mathrm{op}}}$ consists of a sequence of objects X_α and morphisms $\delta^\beta_\alpha \colon X_\beta \to X_\alpha$, for $\alpha < \beta \in \mathbf{Ord}$, defined analogously to Construction 3.5, with $p^\sharp$ and $\mathcal{S}$ replacing p and $\mathcal{M}_{pr}\mathcal{P}_f$.

This terminal sequence converges by the following proposition.

Proposition 4.6. *$\mathcal{S}$ is accessible, and preserves monomorphisms.*

Proof. Since both properties are preserved by composition, it suffices to show that they hold for all the component functors. For $\widehat{\mathcal{M}}_{pr}$, $\widehat{\mathcal{P}_c}$ and $\widehat{\mathcal{P}_f}$, they follow from accessibility and mono-preservation of $\mathcal{M}_{pr}$, $\mathcal{P}_c$ and $\mathcal{P}_f$ (see Proposition 3.6), as (co)limits in presheaf categories are computed pointwise. For $\mathcal{K}$ and $\mathcal{U}$, these properties are proven in [BZ15]. □

Therefore the terminal sequence for $\mathsf{At} \times \mathcal{S}(-)$ converges at some limit ordinal, say γ, yielding the final $\mathsf{At} \times \mathcal{S}(-)$-coalgebra $X_\gamma \xrightarrow{\cong} \mathsf{At} \times \mathcal{S}(X_\gamma)$. The derivation semantics is then

defined $[\![-]\!]_{p^\sharp} \colon \mathsf{At} \to X_\gamma$ by universal property as in the diagram below.

$$\begin{array}{ccc} \mathsf{At} & \xrightarrow{[\![-]\!]_{p^\sharp}} & X_\gamma \\ {\scriptstyle <id_{\mathsf{At}},p^\sharp>}\downarrow & & \downarrow\cong \\ \mathsf{At} \times \mathcal{S}(\mathsf{At}) & \xrightarrow[id_{\mathsf{At}} \times [\![-]\!]_{p^\sharp}]{} & \mathsf{At} \times \mathcal{S}(X_\gamma) \end{array} \tag{4.6}$$

A careful inspection of the terminal sequence constructing X_γ allows to infer a representation of its elements as trees, among which we have those representing computations by unification of goals in a PLP program. We call these *stochastic saturated derivation trees*, as they extend the derivation trees of Definition 3.3 and are the probabilistic variant of saturated and-or trees in [BZ15]. Using (4.6) one can easily verify that $[\![A]\!]_{p^\sharp}$ is indeed the stochastic saturated derivation tree for a given goal A. Example 4.2 provides a pictorial representation of one such tree.

Definition 4.7 (Stochastic saturated derivation trees)**.** Given a probabilistic logic program $\mathbb{P}$, a natural number n and an atom $A \in \mathsf{At}(n)$. The *stochastic saturated derivation tree* for A in $\mathbb{P}$ is the possibly infinite tree $\mathcal{T}$ satisfying the following properties:

(1) There are four kinds of nodes: atom-node (labelled with an atom), substitution-node (labelled with a substitution), clause-node (labelled with $\bullet$), instance-node (labelled with $\blacklozenge$), appearing alternatively in depth in this order. The root is an atom-node with label A.
(2) Each edge between a substitution node and its clause-node children is labelled with some $r \in [0, 1]$.
(3) Suppose an atom-node s is labelled with $A' \in \mathsf{At}(n')$. For every substitution $\theta \colon n' \to m'$, s has exactly one (substitution-node) child t labelled with θ. For every clause $r :: H \leftarrow B_1, \dots, B_k$ in $\mathbb{P}$ such that H matches $A'\theta$ (via some substitution), t has exactly one (clause-)child u, and edge $t \to u$ is labelled with r. Then for every substitution τ such that $A'\theta = H\tau$ and $B_1\tau, \dots, B_k\tau \in \mathsf{At}(m')$, u has exactly one (instance-)child v. Also v has exactly $|\{B_1, \dots, B_k\}|\tau$-many (atom-)children, each labelled with one element in $\{B_1, \dots, B_k\}\tau$.

4.3. **Distribution Semantics.** In this section we conclude by giving a coalgebraic perspective on the distribution semantics $\langle\!\langle - \rangle\!\rangle$ for general PLP. Mimicking the ground case (Section 3.3), this will be presented as an extension of the derivation semantics, via a 'possible worlds' natural transformation. Also in the general case, we want to guarantee that a single probability value is computable for a given goal A from the corresponding tree $\langle\!\langle A \rangle\!\rangle$ in the final coalgebra — whenever this probability is also computable in the 'traditional' way (see (2.1)) of giving distribution semantics to PLP. In this respect, the presence of variables and substitutions poses additional challenges, for which we refer to Appendices A and B. In a nutshell, the issue is that the distribution semantics counts the existence of a clause in the program rather than the number of times a clause is used in the computation. This means that every clause is counted at most once, independently from how many times that clause is used again in the computation. Note that our tree representation, as in the ground case, presents the resolution (thus computation aspect) of the program. So in our tree representation we need to give enough information to determine which clause is used at

each step of the computation, so that a second use can be easily detected. However, neither our saturated derivation trees, nor a 'naive' extension of them to distribution trees, carry such information: what appears in there is only the instantiated heads and bodies, but in general one cannot retrieve A from a substitution θ and the instantiation $A\theta$. This is best illustrated via a simple example.

Example 4.8. Consider the following program, based on the signature $\Sigma = \{a^0\}$ and two 1-ary predicates P, Q. It consists of two clauses:

$$0.5 :: \quad P(x_1) \leftarrow Q(x_1) \qquad \Big| \qquad 0.5 :: \quad P(x_1) \leftarrow Q(x_2)$$

The goal $P(a)$ matches the head of both clauses. However, given the sole information of the next goal being $Q(a)$, it is impossible to say whether the first clause has been used, instantiated with $x_1 \mapsto a$, or the second clause has been used, instantiated with $x_1 \mapsto a, x_2 \mapsto a$.

This observation motivates, as intermediate step towards the distribution semantics, the addition of labels to clause-nodes in derivation trees, in order to make explicit which clause is being applied. From the coalgebraic viewpoint, this just amounts to an extension of the type of the term-matching coalgebra:

$$\widetilde{p} \colon \mathcal{U}\mathsf{At} \to \widehat{\mathcal{M}}_{pr}(\widehat{\mathcal{P}_c}\widehat{\mathcal{P}_f}\mathcal{U}\mathsf{At} \times (\mathcal{U}\mathsf{At} \times \widehat{\mathcal{P}_f}\mathcal{U}\mathsf{At})) \tag{4.7}$$

Note the insertion of $(-) \times (\mathcal{U}\mathsf{At} \times \widehat{\mathcal{P}_f}\mathcal{U}\mathsf{At})$, which allows us to indicate at each step the head ($\mathcal{U}\mathsf{At}$) and the body ($\widehat{\mathcal{P}_f}\mathcal{U}\mathsf{At}$) of the clause being used, its probability label being already given by $\widehat{\mathcal{M}}_{pr}$. More formally, for any n and atom $A \in \mathsf{At}(n)$, we define[1]

$$\widetilde{p}_n(A) \colon \langle \{B_1\tau_i, \ldots, B_k\tau_i\}_{i \in \mathcal{I} \subseteq \mathbb{N}}, \langle H, \{B_1, \ldots, B_k\}\rangle\rangle \mapsto \begin{cases} r & (r :: H \leftarrow B_1, \ldots, B_k) \in \mathbb{P} \\ & \quad \text{and } H\tau_i = A \\ 0 & \text{otherwise} \end{cases}$$

As in the case of p in (4.4), we can move from term-matching to unification by using the universal property of the adjunction $\mathcal{U} \dashv \mathcal{K}$, yielding $\widetilde{p}^\sharp \colon \mathsf{At} \to \mathcal{K}\widehat{\mathcal{M}}_{pr}(\widehat{\mathcal{P}_c}\widehat{\mathcal{P}_f}\mathcal{U}\mathsf{At} \times (\mathcal{U}\mathsf{At} \times \widehat{\mathcal{P}_f}\mathcal{U}\mathsf{At}))$. For simplicity henceforth we denote the functor $\mathcal{K}\widehat{\mathcal{M}}_{pr}(\widehat{\mathcal{P}_c}\widehat{\mathcal{P}_f}\mathcal{U}(-) \times (\mathcal{U}\mathsf{At} \times \widehat{\mathcal{P}_f}\mathcal{U}\mathsf{At}))$ by $\mathcal{R}$.

We are now able to conclude our characterisation of the distribution semantics. The 'possible worlds' transformation $\mathsf{pw} \colon \mathcal{M}_{pr} \Rightarrow \mathcal{D}_{\leq 1}\mathcal{P}_f$ (Definition 3.12) yields a natural transformation $\widehat{\mathsf{pw}} \colon \widehat{\mathcal{M}}_{pr} \to \widehat{\mathcal{D}_{\leq 1}}\widehat{\mathcal{P}_f}$, defined pointwise by pw. We can use $\widehat{\mathsf{pw}}$ to translate $\mathcal{R}$ into the functor $\mathcal{K}\widehat{\mathcal{D}_{\leq 1}}\widehat{\mathcal{P}_f}(\widehat{\mathcal{P}_c}\widehat{\mathcal{P}_f}\mathcal{U}(-) \times (\mathcal{U}\mathsf{At} \times \widehat{\mathcal{P}_f}\mathcal{U}\mathsf{At}))$, abbreviated as $\mathcal{O}$, which is going to give the type of saturated distribution trees for general PLP programs.

As a simple extension of the developments in Section 4.2, we can construct the cofree $\mathcal{R}$-coalgebra $\Phi \xrightarrow{\cong} \mathsf{At} \times \mathcal{R}(\Phi)$ via a terminal sequence. Similarly, one can obtain the cofree $\mathcal{O}$-coalgebra $\Psi \xrightarrow{\cong} \mathsf{At} \times \mathcal{O}(\Psi)$. By the universal property of Ψ, all these ingredients get

[1]As noted in Remark 4.4, instantiating $\widetilde{p}$ to some $n \in \mathsf{Ob}(\mathbf{L}_\Sigma^{\mathrm{op}})$ fixes a variable context $\{x_1, \ldots, x_n\}$ both for the goal and the clause labels. In practice, because the set of clauses is always finite, it suffices to chose n 'big enough' so that the variables appearing in the clauses are included in $\{x_1, \ldots, x_n\}$.

together in the definition of the distribution semantics $\langle\langle - \rangle\rangle_{\widetilde{p}^\sharp}$ for arbitrary PLP programs $\widetilde{p}^\sharp$

$$\begin{array}{ccccc}
\mathsf{At} & \overset{!_\Phi}{\dashrightarrow} & \Phi & \overset{!_\Psi}{\dashrightarrow} & \Psi \\
\downarrow{\scriptstyle <id_{\mathsf{At}},\widetilde{p}^\sharp>} & & \downarrow{\scriptstyle \cong} & & \\
\mathsf{At} \times \mathcal{R}\mathsf{At} & \xrightarrow{id_{\mathsf{At}} \times \mathcal{R}(!_\Phi)} & \mathsf{At} \times \mathcal{R}\Phi & & \downarrow{\scriptstyle \cong} \\
\downarrow{\scriptstyle id_{\mathsf{At}} \times \mathcal{K}\widehat{\mathsf{pw}}} & & \downarrow{\scriptstyle id_{\mathsf{At}} \times \mathcal{K}\widehat{\mathsf{pw}}} & & \\
\mathsf{At} \times \mathcal{O}\mathsf{At} & \xrightarrow{id_{\mathsf{At}} \times \mathcal{O}(!_\Phi)} & \mathsf{At} \times \mathcal{O}\Phi & \xrightarrow{id_{\mathsf{At}} \times \mathcal{O}(!_\Psi)} & \mathsf{At} \times \mathcal{O}\Psi
\end{array}
\qquad \text{(dashed: } \langle\langle - \rangle\rangle_{\widetilde{p}^\sharp} \colon \mathsf{At} \dashrightarrow \Psi\text{)}$$

where $!_\Phi$ and $!_\Psi$ are given by the evident universal properties, and show the role of the cofree $\mathcal{R}$-coalgebra Φ as an intermediate step. The layered construction of final coalgebras Ψ and Φ, together with the above characterisation of $\langle\langle - \rangle\rangle_{\widetilde{p}^\sharp}$, allow to conclude that the distribution semantics for the program $\widetilde{p}^\sharp$ maps a goal A to its *saturated distribution tree* $\langle\langle A \rangle\rangle_{\widetilde{p}^\sharp}$, as formally defined below.

Definition 4.9 (Saturated distribution tree)**.** The *saturated distribution tree* for $A \in \mathsf{At}(n)$ in $\mathbb{P}$ is the possibly infinite $\mathcal{T}$ satisfying the following properties based on Definition 4.7:

(1) There are five kinds of nodes: in addition to the atom-, substitution-, clause- and instance-nodes, there are world-nodes. The world-nodes are children of the substitution-nodes, and parents of the clause nodes. The root and the order of the rest of the nodes are the same as in Definition 4.7, condition **1**. The clause-nodes are now labelled with clauses in $\mathbb{P}$.

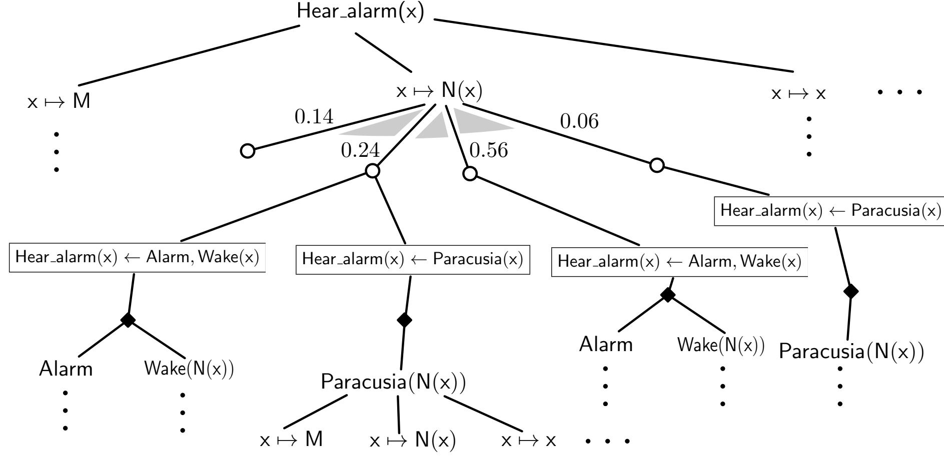
(2) Suppose s is an atom-node labelled with $A' \in \mathsf{At}(n')$, and t is a substitution-child of s labelled with $\theta \colon n' \to m$. Let C be the set of all clauses $\mathcal{C}$ such that $\mathsf{Head}(\mathcal{C})$ matches $A'\theta$. Then t has $2^{|C|}$ world-children, each representing a subset X of C. If a world-child u of t represents subset X, then the edge $t \to u$ has probability label $\prod_{\mathcal{C} \in X} \mathsf{Label}(\mathcal{C}) \cdot \prod_{\mathcal{C}' \in C \setminus X}(1 - \mathsf{Label}(\mathcal{C}'))$. Also u has $|X|$ clause-children, one for each clause $\mathcal{C} \in X$, labelled with the corresponding clause. The rest for clause-nodes and instance-nodes are the same as in Definition 4.7, condition **3**.

Remark 4.10. Note that, in principle, saturated distribution trees could be defined coalgebraically without the intermediate step of adding clause labels. This is to be expected: coalgebra typically captures the one-step, 'local' behaviour of a system. On the other hand, as explained, the need for clause labels is dictated by a computational aspect involving the depth of distribution trees, that is, a 'non-local' dimension of the system.

We conclude with the pictorial representation of the saturated distribution tree of a goal in our leading example.

Example 4.11. In the context of Example 4.1, the tree $\langle\langle \mathsf{Hear_alarm(x)} \rangle\rangle$ capturing the distribution semantics of $\mathsf{Hear_alarm(x)}$ is (partially) depicted as follows. Note the presence

of clauses labelling the clause-nodes.



5. Weighted Logic Programming

Weighted logic programming (WLP) is an abstract framework for describing dynamic programming algorithms in a number of fields, such as natural language processing [EB20] and computational biology [DEKM98]. WLP generalises logic programming by assigning a weight/score to each clause. Given a goal A, the central task then becomes computing the weight of A, based on the weights of the clauses appearing in the proof tree of A.

As WLP and PLP are seemingly close formalisms, it is natural to ask to what extent our coalgebraic perspective applies to WLP. In this section we develop such perspective. By analogy with the development of coalgebraic PLP, we first discuss the ground case (Subsection 5.1), and then move to the general case (Subsection 5.2).

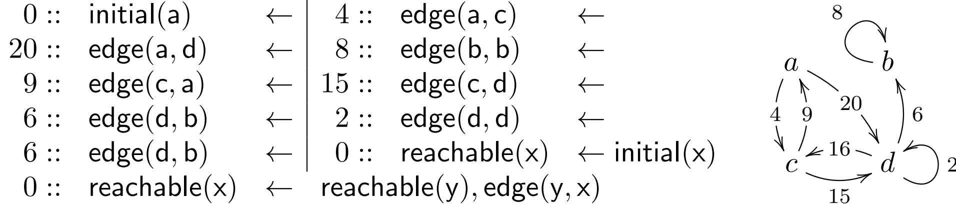
We begin by recalling the basics of WLP, referring to [EB20] for further details. We fix a semiring $\mathbb{K} = \langle K, \oplus, \otimes, \mathbf{0}, \mathbf{1} \rangle$, where $\mathbf{0}$ is the $\oplus$-unit, and $\mathbf{1}$ is the $\otimes$-unit. A weighted logic program $\mathbb{W}$ based on a logic program $\mathbb{L}$ assigns to each clause $\mathcal{C}$ in $\mathbb{L}$ a weight label $c \in K$. In other words, a weighted logic program $\mathbb{W}$ consists of finitely many clauses of the form:

$$c :: H \leftarrow B_1, \dots, B_k$$

where $H, B_1, \dots, B_k$ are atoms and $c \in K$.[2]

Example 5.1. We use as running example the following weighted logic program $\mathbb{P}^{sp}$, from [CSS10]. $\mathbb{P}^{sp}$ is based on the semiring $\mathbb{K}^{sp} = \langle \mathbb{R}_{\geq 0} \cup \{+\infty\}, \min, +, +\infty, 0 \rangle$. Its language consists of constants $\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}$, a 1-ary predicate $\mathsf{initial}(-)$, and 2-ary predicates $\mathsf{edge}(-,-)$ and $\mathsf{reachable}(-,-)$.

[2]It is typical in the literature (e.g. [CSS10]) to only assign weights to facts, i.e. clauses with empty bodies, and leave the other clauses unlabelled. However, we can assume without loss of generality that all clauses are assigned a weight, as the weight semantics (5.1) implicitly assigns weight $\mathbf{1}$ to unlabelled clause.



$\mathbb{P}^{sp}$ describes the single-source directed graph on the right. It expresses that a state x is reachable if either x itself is the initial state, or there is an edge from a reachable state to x. For simplicity, we will sometimes refer to the predicates appearing in $\mathbb{P}^{sp}$ by their abbreviations, such as $\mathsf{init}(-)$, $\mathsf{reach}(-,-)$.

As for the semantics, the central task for WLP is computing the *weight* $\omega^{\mathbb{W}}(A)$ (or simply $\omega(A)$) of an atom A in $\mathbb{W}$. This is an element of $\mathbb{K}$, defined as follows:

$$\omega^{\mathbb{W}}(A) = \bigoplus_{\substack{c::A\leftarrow B_1,\ldots,B_k \text{ is substitution} \\ \text{instance of some clause in } \mathbb{W}}} \left(c \otimes \bigotimes_{B_i} \omega^{\mathbb{W}}(B_i) \right) \tag{5.1}$$

We will refer to (5.1) as the *weight semantics* for WLP. Note that, when the body of a clause is empty, $\bigotimes_{B_i} \omega(B_i)$ is an empty product and yields the unit $\mathbf{1}$. Also, the weight function ω is not necessarily well-defined for every goal and program. For example, consider the program consisting of a single clause $2 :: A \leftarrow A$ with the semiring in Example 5.1. Then (5.1) does not assign a value to A, because the recursive calculation does not terminate.

Example 5.2. In the program $\mathbb{P}^{sp}$ from Example 5.1, the goal $\mathsf{reachable(d)}$ has weight $\omega(\mathsf{reachable(d)}) = 19$, and it is exactly the weight of the shortest path from a to d.

Remark 5.3. One might wonder whether PLP in Section 3 and 4 is a special case of WLP. The answer is negative, for two reasons. First, the underlying algebra structure for PLP is $\langle [0,1], \vee, \times, 0, 1\rangle$, which is not a semiring (distributivity fails). Second, the distribution semantics in (2.1) is not a special case of the weight semantics in (5.1).

5.1. **Coalgebraic Semantics of Ground** WLP**.** We will present the coalgebraic semantics for ground WLP analogous to that for PLP in Section 3: we first give a coalgebraic presentation of WLP; then we induce a derivation semantics for it. The computation of the weight semantics from the derivation semantics is in Appendix C.

A ground WLP program $\mathbb{W}$ can be represented as a coalgebra of the type $\mathcal{MP}_f \colon \mathbf{Sets} \to \mathbf{Sets}$, where $\mathcal{M}$ is the multiset functor corresponding to the underlying semiring $\mathbb{K} = \langle K, \oplus, \otimes, \mathbf{0}, \mathbf{1}\rangle$. On objects, $\mathcal{M} \colon \mathbf{Sets} \to \mathbf{Sets}$ maps a set A to $\{\varphi \colon A \to \mathbb{K} \mid \mathsf{supp}(\varphi) \text{ is finite}\}$, where $\mathsf{supp}(\varphi) = \{a \in A \mid \varphi(a) \neq \mathbf{0}\}$. For each $a \in A$, we call $\varphi(a)$ the weight of a (under φ). We adopt the standard notation $\sum_{i=1}^{k} c_i a_i$ for the function $\varphi \in \mathcal{M}(A)$ with support $\mathsf{supp}(\varphi) = \{a_1, \ldots, a_k\}$ and weights $\varphi(a_i) = c_i$. We also use $\varnothing$ for the multiset with empty support. On morphisms, $\mathcal{M}(h \colon A \to B)$ maps $\sum_{i=1}^{k} c_i a_i$ to $\sum_{i=1}^{k} c_i h(a_i)$.

Fix a ground WLP program $\mathbb{W}$ based on a semiring $\mathbb{K}$ and a set of atoms At. The definition of $\mathbb{W}$ can be encoded as an $\mathcal{MP}_f$-coalgebra $w \colon \mathsf{At} \to \mathcal{MP}_f(\mathsf{At})$ as follows: for

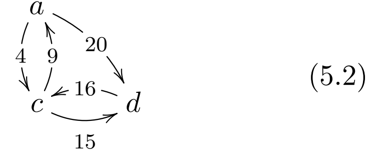
each $A \in \mathsf{At}$,

$$w(A)\colon \quad \mathcal{P}_f(\mathsf{At}) \to K \qquad \{B_1,\ldots,B_n\} \mapsto \begin{cases} c & \text{if } c :: A \leftarrow B_1,\ldots,B_n \text{ is a clause in } \mathbb{W} \\ \mathbf{0} & \text{otherwise.} \end{cases}$$

Example 5.4. Consider the following ground program $\mathbb{P}^{gsp}$ obtained as a simple ground version of $\mathbb{P}^{sp}$. The underlying semiring is still $\mathbb{K}^{sp} = \langle \mathbb{R}_{\geq 0} \cup \{+\infty\}, \min, +, +\infty, 0\rangle$.

$$\begin{array}{rll|rll}
0 :: & \mathsf{initial(a)} & \leftarrow & 0 :: & \mathsf{reachable(d)} & \leftarrow \mathsf{initial(d)} \\
4 :: & \mathsf{edge(a,c)} & \leftarrow & 0 :: & \mathsf{reachable(a)} & \leftarrow \mathsf{reachable(c)}, \mathsf{edge(c,a)} \\
20 :: & \mathsf{edge(a,d)} & \leftarrow & 0 :: & \mathsf{reachable(a)} & \leftarrow \mathsf{reachable(d)}, \mathsf{edge(d,a)} \\
9 :: & \mathsf{edge(c,a)} & \leftarrow & 0 :: & \mathsf{reachable(c)} & \leftarrow \mathsf{reachable(a)}, \mathsf{edge(a,c)} \\
15 :: & \mathsf{edge(c,d)} & \leftarrow & 0 :: & \mathsf{reachable(c)} & \leftarrow \mathsf{reachable(d)}, \mathsf{edge(d,c)} \\
16 :: & \mathsf{edge(d,c)} & \leftarrow & 0 :: & \mathsf{reachable(d)} & \leftarrow \mathsf{reachable(a)}, \mathsf{edge(a,d)} \\
0 :: & \mathsf{reachable(a)} & \leftarrow \mathsf{initial(a)} & 0 :: & \mathsf{reachable(d)} & \leftarrow \mathsf{reachable(c)}, \mathsf{edge(c,d)} \\
0 :: & \mathsf{reachable(c)} & \leftarrow \mathsf{initial(c)} & & &
\end{array}$$

The program describes the (single-sourced, directed) weighted graph (5.2) on the right:



(5.2)

The (finite) set of atoms At_{gsp} is

$$\{\mathsf{initial(a)},\ldots,\mathsf{initial(c)},\mathsf{edge(a,c)},\ldots,\mathsf{edge(d,c)},\mathsf{reachable(a)},\ldots,\mathsf{reachable(c)}\}.$$

Some weights of the corresponding coalgebra $p^{gsp}\colon \mathsf{At}_{gsp} \to \mathcal{M}\mathcal{P}_f(\mathsf{At}_{gsp})$ are presented below:

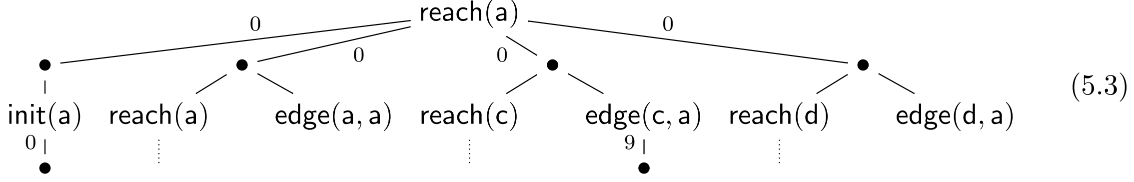
$$p^{gsp}(\mathsf{initial(a)}) = 0\{\} \qquad p^{gsp}(\mathsf{edge(a,d)}) = 20\{\} \qquad p^{gsp}(\mathsf{edge(d,a)}) = \varnothing$$
$$p^{gsp}(\mathsf{reachable(a)}) = 0\{\mathsf{initial(a)}\} + 0\{\mathsf{reachable(c)},\mathsf{edge(c,a)}\} + 0\{\mathsf{reachable(d)},\mathsf{edge(d,a)}\}$$

Note that the 0 in the above equations is the $\otimes$-unit $\mathbf{1}$ in $\mathbb{K}^{sp}$, and should not be confused with the $\oplus$-unit $\mathbf{0}$.

Just as in the case of PLP, we may represent operationally the recursive calculation of the weight $\omega(A)$ of a goal A as a certain kind of tree, which we call *weighted derivation tree*. These are weighted version of the and-or trees for pure logic programming [GC94]. They are formally defined analogously to the stochastic derivation trees in Definition 3.3, except that the probability labels are replaced by weight labels. We illustrate the concept with an example.

Example 5.5. In the context of Example 5.4, the weighted derivation tree for $\mathsf{reachable(a)}$ is partially depicted below. Note the different meaning for an atom-node and a clause-node to have no child. For instance, the clause-node following $\mathsf{init(a)}$ has no child because '$0 :: \mathsf{initial(a)} \leftarrow$' is a clause in $\mathbb{P}^{gsp}$, while the atom-node $\mathsf{edge(d,a)}$ has no child because

there is no clause in $\mathbb{P}^{gsp}$ whose head is $\mathsf{edge}(\mathsf{d},\mathsf{a})$.



(5.3)

By replacing $\mathcal{M}_{pr}$ by $\mathcal{M}$ in Construction 3.5, one may construct the terminal sequence for the functor $\mathsf{At} \times \mathcal{MP}_f(-)$, which by accessibility [Wor99] converges to a limit X_γ, yielding the final $\mathsf{At} \times \mathcal{MP}_f$-coalgebra $X_\gamma \cong \mathsf{At} \times \mathcal{MP}_f(X_\gamma)$ and a unique coalgebra morphism $[\![-]\!]_w$ as below:

$$\begin{array}{ccc} \mathsf{At} & \xrightarrow{\quad [\![-]\!]_w \quad} & X_\gamma \\ \downarrow{\scriptstyle <id,w>} & & \downarrow{\scriptstyle \cong} \\ \mathsf{At} \times \mathcal{MP}_f(\mathsf{At}) & \xrightarrow{id \times \mathcal{MP}_f([\![-]\!]_w)} & \mathsf{At} \times \mathcal{MP}_f(X_\gamma). \end{array}$$

By construction, one may readily verify that weighted derivation trees are elements of X_γ, and $[\![-]\!]_w$ maps an atom A to its weighted derivation tree, computed according to the program $\mathbb{W}$. Thus we can define the derivation semantics for WLP as the map $[\![-]\!]_w$. Note the weight $\omega^w(A)$ of a goal is effectively computable from its semantics $[\![A]\!]_w$ — see Appendix C for details.

5.2. **Coalgebraic Semantics for Arbitrary WLP Programs.** In this subsection we generalise our approach to arbitrary weighted logic programs, possibly including variables and functions. We shall assume that the semiring $\mathbb{K}$ underlying the WLP programs is countably $\oplus$-complete, namely $\left(\bigoplus_{c\in C} c\right) \in K$ and $d \otimes \left(\bigoplus_{c\in C} c\right) = \bigoplus_{c\in C}(d \otimes c)$, for arbitrary countable subset $C \subseteq K$. As we will see, the reason for such assumption is that there may exist countably many substitutions that match the head of some clause to the goal, generating countably many different instances of the body. The weight semantics requires adding up all these countably many weights, whence their sum should be an element of $\mathbb{K}$.

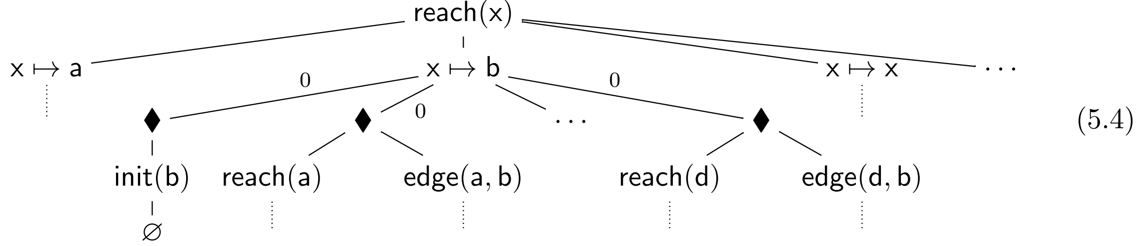
The construction leading to a coalgebraically defined derivation semantics $[\![-]\!]$ for WLP is fairly similar to the one for PLP (Subsection 4.2), thus we confine ourselves to highlighting the main steps and emphasise the differences between the two approaches.

The derivation semantics $[\![-]\!]$ will associate a goal with its resolution by unification in a program. This resolution will be represented as a tree, which we call *weighted saturated derivation tree.* Formally, these are defined analogously to stochastic saturated derivation trees (Definition 4.7), with the difference that there is no need of clause-nodes (for reasons detailed in Remark 5.7 below), and labels on edges departing from substitution-nodes are now weights rather than probabilities.

Before providing a coalgebraic modelling of the semantics, we establish some preliminary intuition via our leading example.

Example 5.6. Recall program $\mathbb{P}^{sp}$ from Example 5.1. We show below part of the weighted saturated derivation tree for the goal $\mathsf{reachable}(\mathsf{x})$ (in context 1), where we write $\varnothing$ for the empty substitution. Compared with the ground case (see (5.3) in Example 5.5), the tree in (5.4) has an extra layer of 'substitution-nodes', labelled with the substitutions applied to the goal during resolution by unification. For instance, the node labelled with $\mathsf{x} \mapsto \mathsf{b}$ has one child for each substitution instance of the body $\{B_1, \ldots, B_k\}\tau$ satisfying that $c :: H \leftarrow B_1, \ldots, B_k$

is a clause in $\mathbb{P}^{sp}$ and the head H term-matches the goal $\mathsf{reach}(\mathsf{x})[\mathsf{x} \mapsto \mathsf{b}] = \mathsf{reach}(\mathsf{b})$ via substitution τ.



(5.4)

As mentioned, with respect to stochastic saturated derivation trees (see e.g. (4.1)), the semantics represented in (5.4) does not require the presence of clause-nodes (which were labelled with $\bullet$ in (4.1)).

Similarly to the case of PLP, we will describe resolution by unification in two steps: first, one considers all substitutions θ of the goal A. Then, each substitution instance $A\theta$ of the goal is compared by term-matching to the heads of clauses in the program $\mathbb{W}$. The function u performing term-matching of $A\theta$ in $\mathbb{W}$ associates a set $\{C_1, \ldots, C_m\}$ of atoms with an element of $\mathbb{K}$, computed as follows:

$$u(A\theta)\colon \{C_1, \ldots, C_m\} \mapsto \bigoplus_{\substack{c::H\leftarrow B_1,\ldots,B_k \text{is a clause of } \mathbb{W} \\ \text{such that } \exists\tau \text{ with } H\tau=A\theta \\ \text{and } \{B_1,\ldots,B_k\}\tau=\{C_1,\ldots,C_m\}}} c \tag{5.5}$$

Intuitively, for each clause $c :: H \leftarrow B_1, \ldots, B_k$ in $\mathbb{W}$ and substitution τ that matches the head H with $A\theta$, $u(A\theta)$ assigns weight c to $\{B_1, \ldots, B_k\}\tau$. There are three observations to make here. First, it is possible that there is a different clause $c' :: H' \leftarrow B'_1, \ldots, B'_{k'}$ and substitution τ' such that $H'\tau' = A\theta$, and $\{B'_1, \ldots, B'_{k'}\}\tau' = \{B_1, \ldots, B_k\}\tau = \{C_1, \ldots, C_m\}$. In this case, the weight c' is also added to the value of $u(A\theta)(\{C_1, \ldots, C_m\})$. Second, there may exist a different substitution σ such that $H\tau = H\sigma$ and $\{B_1, \ldots, B_k\}\tau = \{B_1, \ldots, B_k\}\sigma$. In this case, no extra weight c is added to $u(A\theta)(\{B_1, \ldots, B_k\}\tau)$. Third, the support $\mathsf{supp}(u(A\theta))$ of the function might be countably infinite. This is due to the possible existence of a clause $H \leftarrow B_1, \ldots, B_k$ and countably many substitutions $\tau_1, \tau_2, \ldots$ such that $H\tau_i = A\theta$ while $\{B_1, \ldots, B_k\}\tau_i \neq \{B_1, \ldots, B_k\}\tau_j$, for all distinct $i, j = 1, 2, \ldots$.

For example, the application of u to $\mathsf{reach}(\mathsf{b})$ in Example 5.6 can be visualised in tree form as the first two layers of the subtree starting at $\mathsf{x} \mapsto \mathsf{b}$.

Towards a categorical treatment of resolution by unification, we need to introduce the *countable* (because of the third observation above) multiset functor $\mathcal{M}_c\colon \mathbf{Sets} \to \mathbf{Sets}$ over a $\oplus$-complete semiring $\mathbb{K}$. On objects, $\mathcal{M}_c(X) = \{\varphi\colon X \to \mathbb{K} \mid \mathsf{supp}(\varphi) \text{ is at most countable}\}$. On morphisms, $\mathcal{M}_c(h)\colon \mathcal{M}_c(X) \to \mathcal{M}_c(Y)$ maps φ to $\lambda(y \in Y). \bigoplus_{x\in h^{-1}(y)} \varphi(x)$. Now, lifting functors from $\mathbf{Sets}$ to $\mathbf{Sets}^{|\mathbf{L}_\Sigma^{\mathrm{op}}|}$ as in (4.4), the term-matching part u of the resolution can be expressed as a $\widehat{\mathcal{M}_c}\widehat{\mathcal{P}_f}$-coalgebra in $\mathbf{Sets}^{|\mathbf{L}_\Sigma^{\mathrm{op}}|}$:

$$\mathsf{u}\colon \mathcal{U}\mathsf{At} \to \widehat{\mathcal{M}_c}\widehat{\mathcal{P}_f}\mathcal{U}\mathsf{At} \tag{5.6}$$

Remark 5.7. Note the coalgebra type of term-matching for WLP ($\widehat{\mathcal{M}_c}\widehat{\mathcal{P}_f}$) is simpler than the one for PLP ($\widehat{\mathcal{M}}_{pr}\widehat{\mathcal{P}_c}\widehat{\mathcal{P}_f}$), as in (4.4). In fact, in the distribution semantics of PLP, the use of a clause in a proof should be counted at most once. This is taken care by the extra layer $\widehat{\mathcal{M}}_{pr}$ which allows to identify different substitution instances of the same clause in a single

tree. Instead, in the weight semantics every use of a clause counts towards calculating the total weight (*cf.* (5.1)). Thus it is unnecessary to keep track of whether clauses are used more than once.

This difference is reflected in the tree representation of the two semantics, compare e.g. (5.4) with (4.1). In the weighted saturated trees, there is no need for clause-nodes, which are instead present in stochastic saturated trees.

Analogously to the PLP case, once term matching is modelled as a coalgebra, we can then exploit the unit η of the adjunction $\mathcal{U} \dashv \mathcal{K}$ (see (4.3)) to model the retrieval of all the substitutions of the goals. This will yield a $\mathcal{K}\widehat{\mathcal{M}_c}\widehat{\mathcal{P}_f}\mathcal{U}$-coalgebra in $\mathbf{Sets}^{\mathbf{L}_\Sigma^{\mathrm{op}}}$, capturing resolution by unification:

$$\mathsf{u}^\sharp := \mathsf{At} \xrightarrow{\eta_{\mathsf{At}}} \mathcal{K}\mathcal{U}\mathsf{At} \xrightarrow{\mathcal{K}\mathsf{u}} \mathcal{K}\widehat{\mathcal{M}_c}\widehat{\mathcal{P}_f}\mathcal{U}\mathsf{At} \tag{5.7}$$

Spelling out the definitions, given some $A \in \mathsf{At}(n)$,

$$\mathsf{u}^\sharp_n \colon A \mapsto \langle \mathsf{u}_m(A\theta)\rangle_{\theta\in\mathbf{L}_\Sigma^{\mathrm{op}}[n,m]}$$

Finally, we can formulate the derivation semantics for general WLP via the cofree coalgebra. First, one may generate the terminal sequence for the functor $\mathsf{At} \times \mathcal{K}\widehat{\mathcal{M}_c}\widehat{\mathcal{P}_f}\mathcal{U}$, following a similar procedure as Construction 4.5. Since $\mathcal{M}_c$ is accessible, the terminal sequence converges, say at Z_γ. Then the derivation semantics $[\![-]\!]_{\mathsf{u}^\sharp}$ is defined by the universal mapping property of the final coalgebra:

$$\begin{array}{ccc}
\mathsf{At} & \xrightarrow{[\![-]\!]_{\mathsf{u}^\sharp}} & Z_\gamma \\
\Big\downarrow {\scriptstyle <id,\mathsf{u}^\sharp>} & & \Big\downarrow \cong \\
\mathsf{At} \times \mathcal{K}\widehat{\mathcal{M}_c}\widehat{\mathcal{P}_f}\mathcal{U}\mathsf{At} & \xrightarrow[id\times\mathcal{K}\widehat{\mathcal{M}_c}\widehat{\mathcal{P}_f}\mathcal{U}[\![-]\!]_{\mathsf{u}^\sharp}]{} & \mathsf{At} \times \mathcal{K}\widehat{\mathcal{M}_c}\widehat{\mathcal{P}_f}\mathcal{U}Z_\gamma
\end{array} \tag{5.8}$$

As expected, weighted saturated derivation trees can be regarded as elements of the final coalgebra Z_γ, and $[\![-]\!]_{\mathsf{u}^\sharp}$ maps an atom A to its weighted saturated derivation tree.

6. Conclusion

This work proposed a coalgebraic semantics for probabilistic logic programming and weighted logic programming, as an extension of the framework in [KMP10, BZ15] for pure logic programming. Our approach consists of the following three steps. First we represent the programs as coalgebras of appropriate type. Next we define the derivation semantics in terms of certain derivation trees, which can be recovered as objects in the corresponding final coalgebra. Finally, we explain how to retrieve the semantics that is most commonly found in the literature (distribution semantics for PLP and weight semantics for WLP) from our coalgebraic semantics. While the first two steps simply generalise the approach in [BZ15], the last step has no counterpart in pure logic programming, and it requires extra categorical constructions (distribution trees) and computations (Appendix A, B, C).

In extending our coalgebraic semantics from ground to arbitrary PLP programs, we adopted the 'saturated' approach of [BZ15]. The 'lax naturality' approach of [KP11] offers an alternative route (see [KP18a] for a detailed comparison), based on the observation that, even though first-order logic programs fail to be natural transformations, they are lax natural in a suitable order enriched category. Developing a coalgebraic lax semantics for PLP and WLP would yield derivation trees representing computations by term-matching

rather than general unification. The extra information given by lax naturality could be used for algorithmic purposes, similarly to what is done in [KSH14, KPS13] for pure logic programming. We leave the development of lax semantics for PLP and WLP as future work.

As follow-up work, we plan to investigate in two main directions. First, starting from the observation that Bayesian networks (over binary variables) can be seen as PLP programs, we want to understand Bayesian reasoning within our coalgebraic framework. Recent works on a categorical semantics for Bayesian probability (in particular [JZ19, CJ19]) provide a starting point for this research. A major challenge is that standard translations of Bayesian networks into PLP (such as the one in [Poo08]) require to incorporate negation in the logical syntax. Thus a preliminary step will be to model coalgebraically PLP with negation, which we plan to do following the credal set approach outlined in [Coz19].

A second research thread is investigating the semantics of coinductive logic programs in the quantitative setting of PLP and WLP. This is an essentially unexplored perspective, which we will tackle building on research on coinduction in the coalgebraic semantics of standard logic programming [KL18, GBM+07, BKL19].

References

[BKL19] Henning Basold, Ekaterina Komendantskaya, and Yue Li. Coinduction in uniform: Foundations for corecursive proof search with horn clauses. Lecture Notes in Computer Science, pages 783–813. Springer, 2019.

[BSdV04] Falk Bartels, Ana Sokolova, and Erik P. de Vink. A hierarchy of probabilistic system types. *Theor. Comput. Sci.*, 327(1-2):3–22, 2004.

[BZ13] Filippo Bonchi and Fabio Zanasi. Saturated semantics for coalgebraic logic programming. In *Algebra and Coalgebra in Computer Science - 5th International Conference, CALCO 2013, Warsaw, Poland, September 3-6, 2013. Proceedings*, pages 80–94, 2013.

[BZ15] Filippo Bonchi and Fabio Zanasi. Bialgebraic semantics for logic programming. *Logical Methods in Computer Science*, 11(1), 2015.

[CJ19] Kenta Cho and Bart Jacobs. Disintegration and bayesian inversion via string diagrams. *Mathematical Structures in Computer Science*, 29(7):938–971, 2019.

[Coz19] Fabio Cozman. The joy of probabilistic answer set programming. In Jasper De Bock, Cassio P. de Campos, Gert de Cooman, Erik Quaeghebeur, and Gregory Wheeler, editors, *Proceedings of the Eleventh International Symposium on Imprecise Probabilities: Theories and Applications*, volume 103 of *Proceedings of Machine Learning Research*, pages 91–101, Thagaste, Ghent, Belgium, 03–06 Jul 2019. PMLR.

[CSS10] Shay Cohen, Robert Simmons, and Noah Smith. Products of weighted logic programs. *Computing Research Repository - CORR*, 11, 06 2010.

[Dan92] Eugene Dantsin. Probabilistic logic programs and their semantics. In A. Voronkov, editor, *Logic Programming*, pages 152–164, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

[DDGK16] Fredrik Dahlqvist, Vincent Danos, Ilias Garnier, and Ohad Kammar. Bayesian inversion by ω-complete cone duality. In *27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada*, pages 1:1–1:15, 2016.

[DEKM98] Richard Durbin, Sean R. Eddy, Anders Krogh, and Graeme J. Mitchison. Biological sequence analysis: Probabilistic models of proteins and nucleic acids. Cambridge University Press, 1998.

[DRKT07a] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. Problog: A probabilistic prolog and its application in link discovery. In *Proceedings of the 20th International Joint Conference on Artifical Intelligence*, IJCAI'07, pages 2468–2473, San Francisco, CA, USA, 2007. Morgan Kaufmann Publishers Inc.

[DRKT07b] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. Problog: A probabilistic prolog and its application in link discovery. In *Proceedings of the 20th International Joint Conference on Artifical Intelligence*, IJCAI'07, pages 2468–2473, San Francisco, CA, USA, 2007. Morgan Kaufmann Publishers Inc.

[EB20] Jason Eisner and John Blatz. Program transformations for optimization of parsing algorithms and other weighted logic programs. 02 2020.

[GBM+07] Gopal Gupta, Ajay Bansal, Richard Min, Luke Simon, and Ajay Mallya. Coinductive logic programming and its applications. In Véronica Dahl and Ilkka Niemelä, editors, *Logic Programming*, pages 27–44, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[GC94] Gopal Gupta and Vítor Santos Costa. Optimal implementation of and-or parallel prolog. *Future Generation Computer Systems*, 10(1):71 – 92, 1994. PARLE '92.

[GZ19] Tao Gu and Fabio Zanasi. A coalgebraic perspective on probabilistic logic programming. In *8th Conference on Algebra and Coalgebra in Computer Science, CALCO 2019, June 3-6, 2019, London, United Kingdom*, pages 10:1–10:21, 2019.

[JKZ19] Bart Jacobs, Aleks Kissinger, and Fabio Zanasi. Causal inference by string diagram surgery. In *Foundations of Software Science and Computation Structures - 22nd International Conference, FOSSACS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Proceedings*, 2019.

[JZ19] Bart Jacobs and Fabio Zanasi. The logical essentials of bayesian reasoning. In Joost-Peter Katoen Gilles Barthe and Alexandra Silva, editors, *Probabilistic Programming*. Cambridge University Press, Cambridge, 2019.

[KL17] Ekaterina Komendantskaya and Yue Li. Productive corecursion in logic programming. *TPLP*, 17(5-6):906–923, 2017.

[KL18] Ekaterina Komendantskaya and Yue Li. Towards coinductive theory exploration in horn clause logic: Position paper. In *Proceedings 5th Workshop on Horn Clauses for Verification and Synthesis, HCVS 2018, Oxford, UK, 13th July 2018.*, pages 27–33, 2018.

[KMP10] Ekaterina Komendantskaya, Guy McCusker, and John Power. Coalgebraic semantics for parallel derivation strategies in logic programming. In *Algebraic Methodology and Software Technology - 13th International Conference, AMAST 2010, Lac-Beauport, QC, Canada, June 23-25, 2010. Revised Selected Papers*, pages 111–127, 2010.

[KP11] Ekaterina Komendantskaya and John Power. Coalgebraic semantics for derivations in logic programming. In *Algebra and Coalgebra in Computer Science - 4th International Conference, CALCO 2011, Winchester, UK, August 30 - September 2, 2011. Proceedings*, pages 268–282, 2011.

[KP18a] Ekaterina Komendantskaya and John Power. Logic programming: laxness and saturation. *Journal of logical and algebraic methods in programming*, 101:1–21, 2018.

[KP18b] Ekaterina Komendantskaya and John Power. Logic programming: Laxness and saturation. *J. Log. Algebr. Meth. Program.*, 101:1–21, 2018.

[KPS13] Ekaterina Komendantskaya, John Power, and Martin Schmidt. Coalgebraic logic programming: from semantics to implementation. *CoRR*, abs/1312.6568, 2013.

[KPS16] Ekaterina Komendantskaya, John Power, and Martin Schmidt. Coalgebraic logic programming: from semantics to implementation. *J. Log. Comput.*, 26(2):745–783, 2016.

[KSH14] Ekaterina Komendantskaya, Martin Schmidt, and Jónathan Heras. Exploiting parallelism in coalgebraic logic programming. *Electron. Notes Theor. Comput. Sci.*, 303:121–148, 2014.

[Llo87] John W. Lloyd. *Foundations of Logic Programming, 2nd Edition*. Springer, 1987.

[Mac71] Saunders MacLane. *Categories for the Working Mathematician*. Springer-Verlag, 1971. Graduate Texts in Mathematics, Vol. 5.

[MH12] Søren Mørk and Ian Holmes. Evaluating bacterial gene-finding hmm structures as probabilistic logic programs. *Bioinformatics*, 28(5):636–642, 2012.

[NS92] Raymond Ng and V.S. Subrahmanian. Probabilistic logic programming. *Information and Computation*, 101(2):150 – 201, 1992.

[Poo08] David Poole. The independent choice logic and beyond. In Luc De Raedt, Paolo Frasconi, Kristian Kersting, and Stephen Muggleton, editors, *Probabilistic Inductive Logic Programming - Theory and Applications*, volume 4911 of *Lecture Notes in Computer Science*, pages 222–243. Springer, 2008.

[RS14] Fabrizio Riguzzi and Terrance Swift. Probabilistic logic programming under the distribution semantics. 2014.

[Sat95] Taisuke Sato. A statistical learning method for logic programs with distribution semantics. In *Logic Programming, Proceedings of the Twelfth International Conference on Logic Programming, Tokyo, Japan, June 13-16, 1995*, pages 715–729, 1995.

[Sil10] Alexandra Silva. Kleene coalgebra. Phd thesis, CWI, Amsterdam, The Netherlands, 2010.

[TBF05] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic robotics*. MIT Press, Cambridge, Mass., 2005.

[Wor99] James Worrell. Terminal sequences for accessible endofunctors. *Electronic Notes in Theoretical Computer Science*, 19:24 – 38, 1999. CMCS'99, Coalgebraic Methods in Computer Science.

[Zes17] Riccardo Zese. *Probabilistic Semantic Web: Reasoning and Learning*. IOS Press, Amsterdam, The Netherlands, The Netherlands, 2017.

Appendix A. Computability of the Distribution Semantics (Ground Case)

Computing with distribution trees. As a justification for our tree representation of the distribution semantics, we claimed that the probability $\mathsf{Pr}_{\mathbb{P}}(A)$ associated with a goal (see (2.1)) can be straightforwardly computed from the corresponding distribution tree $\langle\!\langle A \rangle\!\rangle_p$. This appendix supplies such an algorithm. Note that this serves just as a proof of concept, without any claim of efficiency compared to pre-existing implementations. In this section we fix a ground PLP program $\mathbb{P}$ with atoms At, a goal $A \in \mathsf{At}$ and the distribution tree $\mathcal{T}$ for A in $\mathbb{P}$ (Definition 3.8). First, we may assume that the distribution tree $\mathcal{T}$ is finite. Indeed, in the ground case infinite branches only result from loops, namely multiple appearances of an atom in some path, which can be easily detected. We can always prune the subtrees of $\mathcal{T}$ rooted by atoms that already appeared at an earlier stage: this does not affect the computation of $\mathsf{Pr}_{\mathbb{P}}(A)$, and it makes $\mathcal{T}$ finite. Next, we introduce the concept of *deterministic* subtree. Basically a deterministic subtree selects one world-node at each stage. Recall that every clause-node in $\mathcal{T}$ represents a clause in $|\mathbb{P}|$, whose head is the label of its atom-grandparent, and body consists of the labels of its atom-children.

Definition A.1. A subtree $\mathcal{S}$ of $\mathcal{T}$ is *deterministic* if (i) it contains exactly one child (world-node) for each atom-node and all children for other nodes, and (ii) for any distinct atom-nodes s, t in $\mathcal{S}$ with the same label, s and t have their clause-grandchildren representing the same clauses.

The idea is that $\mathcal{S}$ describes a computation in which the choice of a possible world (i.e., a sub-program of $\mathbb{P}$) associated to any atom B appearing during the resolution is uniquely determined. Because of this feature, each deterministic subtree uniquely identifies a set of sub-programs of $\mathbb{P}$, and together the deterministic subtrees of $\mathcal{T}$ form a *partition* over the set of these sub-programs (see Proposition A.3 below).

Since $\mathcal{T}$ is finite, it is clear that we can always provide an enumeration of its deterministic subtrees. We can now present our algorithm, in two steps. First, Algorithm 1 computes the probability associated with a deterministic subtree. Second, Algorithm 2 computes $\mathsf{Pr}_{\mathbb{P}}(A)$ by summing up the probabilities found by Algorithm 1 on all the deterministic subtrees of $\mathcal{T}$ that contains a refutation of A. Below we write 'label($s \to t$)' for the probability value labelling the edge from s to t.

Algorithm 1 Compute probability of a deterministic subtree

Input: A deterministic subtree $\mathcal{S}$ of $\mathcal{T}$
Output: The probability of $\mathcal{S}$

```
1: prob_list = [ ]
2: for atom-node s in S do
3:     if s has child then
4:         prob_list.append(label(s → child(s)))
5: if prob_list == [ ] then
6:     return 0
7: else prob = product of values in prob_list
8:     return prob
```

Algorithm 2 Compute probability of a goal

Input: The distribution tree $\mathcal{T}$ of A in $\mathbb{P}$
Output: The success probability $\mathsf{Pr}_{\mathbb{P}}(A)$

```
1: prob_suc = 0
2: for deterministic subtree S of T do
3:     if S proves A then
4:         prob_suc += Algorithm 1(S)
5: return prob_suc
```

The above procedure terminates because $\mathcal{T}$ is finite and every for-loop is finite. We now focus on the correctness of the algorithm.

Correctness. As mentioned, a world-node in a deterministic subtree can be seen as a choice of clauses: one chooses the clauses represented by its clause-children, and discards the clauses represented by its 'complement' world. For correctness, we make this precise, via the following definition.

Definition A.2. Given a clause $\mathcal{C}$ in $\mathbb{P}$, a deterministic subtree $\mathcal{S}$ of $\mathcal{T}$, a world-node t and its atom-parent s in $\mathcal{S}$, we say t *accepts* $\mathcal{C}$ if $\mathsf{Head}(\mathcal{C}) = \text{label}(s)$ and there is a clause-child of t that represents $\mathcal{C}$; t *rejects* $\mathcal{C}$ if $\mathsf{Head}(\mathcal{C}) = \text{label}(s)$ but no clause-child of t represents $\mathcal{C}$. We say $\mathcal{S}$ *accepts* (*rejects*) $\mathcal{C}$ if there exists a world-node t in $\mathcal{S}$ that accepts (rejects) $\mathcal{C}$.

Note that Definition A.1, condition (ii) prevents the existence of world-nodes t, t' in $\mathcal{S}$ such that t accepts $\mathcal{C}$ and t' rejects $\mathcal{C}$. Thus the notion that $\mathcal{S}$ accepts (rejects) $\mathcal{C}$ is well-defined. We denote the set of clauses accepted and rejected by $\mathcal{S}$ by $\mathsf{Acc}(\mathcal{S})$ and $\mathsf{Rej}(\mathcal{S})$, respectively. Then we can define the set $\mathsf{SubProg}(\mathcal{S})$ of sub-programs represented by $\mathcal{S}$ as

$$\mathsf{SubProg}(\mathcal{S}) \coloneqq \{\mathbb{L} \subseteq |\mathbb{P}| \mid \forall \mathcal{C} \in \mathsf{Acc}(\mathcal{S}), \mathcal{C} \in \mathbb{L}; \forall \mathcal{C}' \in \mathsf{Rej}(\mathcal{S}), \mathcal{C}' \notin \mathbb{L}\} \tag{A.1}$$

We will prove the correctness of the algorithm through the following basic observations on the connection between deterministic subtrees and the sub-programs they represent:

Proposition A.3. *Suppose $\mathcal{S}$ is a deterministic subtree of the distribution tree $\mathcal{T}$ of A.*

(1) $\{\mathsf{SubProg}(\mathcal{S}) \mid \mathcal{S}$ *is deterministic subtree of* $\mathcal{T}\}$ *forms a partition of* $\mathcal{P}(\mathbb{P})$*.*
(2) *Either* $\mathbb{L} \vdash A$ *for all* $\mathbb{L} \in \mathsf{SubProg}(\mathcal{S})$ *or* $\mathbb{L} \nvdash A$ *for all* $\mathbb{L} \in \mathsf{SubProg}(\mathcal{S})$*.*
(3) $\sum_{\mathbb{L} \in \mathsf{SubProg}(\mathcal{S})} \mathsf{Pr}_{\mathbb{P}}(\mathbb{L}) = \prod_{r_i \in \mathcal{S}} r_i$*, where the* r_i*s are all the probability labels appearing in* $\mathcal{S}$ *(on the atom-node* $\to$ *world-node edges).*

Proof.

(1) Given any two distinct deterministic subtrees, there is an atom-node s such that the subtrees include distinct world-child of s. So by (A.1) the sub-programs they represent do not share at least one clause. Moreover, given a sub-program $\mathbb{L}$, one can always identify a deterministic subtree $\mathcal{S}$ such that $\mathbb{L} \in \mathsf{SubProg}(\mathcal{S})$, as follows: given the A-labelled root of $\mathcal{T}$, select the world-child w of A representing the (possibly empty) set X of all clauses in $\mathbb{L}$ whose head is A; then select the children (if any) of w, and repeat the procedure.
(2) Note that a sub-program $\mathbb{L} \in \mathsf{SubProg}(\mathcal{S})$ proves the goal A iff $\mathcal{S}$ contains a successful refutation of A, and the latter property is independent of the choice of $\mathbb{L}$.

(3) We refer to $\prod_{r_i \in \mathcal{S}} r_i$ as the probability of the deterministic subtree $\mathcal{S}$. For each subprogram $\mathbb{L} \in \mathsf{SubProg}(\mathcal{S})$, its probability can be written as

$$\mu_{\mathbb{P}}(\mathbb{L}) = \left(\prod_{\mathcal{C} \in \mathsf{Acc}(\mathcal{S})} \mathsf{Label}(\mathcal{C}) \right) \cdot \left(\prod_{\mathcal{C}' \in \mathsf{Rej}(\mathcal{S})} \big(1 - \mathsf{Label}(\mathcal{C}')\big) \right) \cdot \mu_{\mathbb{P} \setminus (\mathsf{Acc} \cup \mathsf{Rej})}(\mathbb{L} \setminus \mathsf{Acc}(\mathcal{S})) \quad \text{(A.2)}$$

Note that $\mathsf{SubProg}(\mathcal{S})$ can also be written as $\{X \cup \mathsf{Acc}(\mathcal{S}) \mid X \subseteq \mathbb{P} \setminus (\mathsf{Acc}(\mathcal{S}) \cup \mathsf{Rej}(\mathcal{S}))\}$, so

$$\sum_{\mathbb{L} \in \mathsf{SubProg}(\mathcal{S})} \mu_{\mathbb{P} \setminus (\mathsf{Acc} \cup \mathsf{Rej})}(\mathbb{L} \setminus \mathsf{Acc}(\mathcal{S})) = 1. \quad \text{(A.3)}$$

Applying equation (A.3) to the sum of (A.2) over all $\mathbb{L} \in \mathsf{SubProg}(\mathcal{S})$, we get

$$\sum_{\mathbb{L} \in \mathsf{SubProg}(\mathcal{S})} \mu_{\mathbb{P}}(\mathbb{L}) = \prod_{\mathcal{C} \in \mathsf{Acc}(\mathcal{S})} \mathsf{Label}(\mathcal{C}) \cdot \prod_{\mathcal{C}' \in \mathsf{Rej}(\mathcal{S})} (1 - \mathsf{Label}(\mathcal{C}')) \quad \text{(A.4)}$$

For each world-node t and its atom-parent s, we can use the terminology in Definition A.2, and express label$(s \to t)$ (see Definition 3.8) as

$$\text{label}(s \to t) = \prod_{t \text{ accepts } \mathcal{C}} \mathsf{Label}(\mathcal{C}) \cdot \prod_{t \text{ rejects } \mathcal{C}'} (1 - \mathsf{Label}(\mathcal{C}')). \quad \text{(A.5)}$$

Applying (A.5) to the whole deterministic subtree $\mathcal{S}$, we obtain

$$\begin{aligned}
\sum_{\mathbb{L} \in \mathsf{SubProg}(\mathcal{S})} \mu_{\mathbb{P}}(\mathbb{L}) &\overset{\text{(A.4)}}{=} \prod_{\mathcal{C} \in \mathsf{Acc}(\mathcal{S})} \mathsf{Label}(\mathcal{C}) \cdot \prod_{\mathcal{C}' \in \mathsf{Rej}(\mathcal{S})} (1 - \mathsf{Label}(\mathcal{C}')) \\
&\overset{\text{Def.}A.2}{=} \prod_{(\text{world-node } t \text{ in } \mathcal{S})} \left(\prod_{t \text{ accepts } \mathcal{C}} \mathsf{Label}(\mathcal{C}) \cdot \prod_{t \text{ rejects } \mathcal{C}'} (1 - \mathsf{Label}(\mathcal{C}')) \right) \\
&\overset{\text{(A.5)}}{=} \prod_{r_i \in \mathcal{S}} r_i
\end{aligned}$$

If we say two world-nodes t and t' are equivalent if their clause-children represent exactly the same clauses in $\mathbb{P}$, then the $\prod_{(\text{world-node } t \text{ in } \mathcal{S})}$ in the above calculation visits every world-node exactly once modulo equivalence. □

We can now formulate the success probability of A as follows

$$\begin{aligned}
\mathsf{Pr}_{\mathbb{P}}(A) = \sum_{|\mathbb{P}| \supseteq \mathbb{L} \vdash A} \mu_{\mathbb{P}}(\mathbb{L}) &\overset{(\text{Prop.}A.3,1\&2)}{=} \sum_{\mathcal{S} \vdash A} \sum_{\mathbb{L} \in \mathsf{SubProg}(\mathcal{S})} \mu_{\mathbb{P}}(\mathbb{L}) \\
\overset{\text{(A.4)}}{=} \sum_{\mathcal{S} \vdash A} \left(\prod_{\mathcal{C} \in \mathsf{Acc}(\mathcal{S})} \mathsf{Label}(\mathcal{C}) \cdot \prod_{\mathcal{C}' \in \mathsf{Rej}(\mathcal{S})} (1 - \mathsf{Label}(\mathcal{C}')) \right) &\overset{(\text{Prop.}A.3,3)}{=} \sum_{\mathcal{S} \vdash A} \prod_{r_i \in S} r_i
\end{aligned}$$

In words, this is exactly Algorithm 2: we sum up the probabilities of all deterministic subtrees $\mathcal{S}$ of the distribution tree $\mathcal{T}$ which contain a proof of A.

Appendix B. Computability of the Distribution Semantics (General Case)

Computability of the distribution semantics for arbitrary PLP programs relies on the substitution mechanism employed in the resolution. This aspect deserves a preliminary discussion. Traditionally, logic programming has both the theorem-proving and problem-solving perspectives [KP18b]. From the problem-solving perspective, the aim is to find a refutation of the goal $\leftarrow G$, which amounts to finding a proof of *some substitution instance* of G. From the theorem-proving perspective, the aim is to search for a proof of the goal G itself as an atom. The main difference is in the substitution mechanism of resolution: unification for the problem-solving and term-matching for the theorem-proving perspective. We will first explore computability within the theorem-proving perspective. As resolution therein is by term-matching, the probability $\mathsf{Pr}^{\mathsf{TM}}_{\mathbb{P}}(A)$ of proving a goal A in a PLP program $\mathbb{P}$ is formulated as $\mathsf{Pr}^{\mathsf{TM}}_{\mathbb{P}}(A) := \sum_{|\mathbb{P}| \supseteq \mathbb{L} \Rightarrow A} \mu_{\mathbb{P}}(\mathbb{L})$, where $\mathbb{L} \Rightarrow A$ means that A is derivable in the sub-program $\mathbb{L}$ (not to be confused with $\mathbb{L} \vdash A$, which stands for *some substitution instance* of A being derivable in $\mathbb{L}$, see (2.1)).

In our coalgebraic framework, the distribution semantics for general PLP programs is represented on 'saturated' trees, in which computations are performed by unification. However, following [BZ15], one can define the *TM (**T**erm **M**atching) distribution tree* of a goal A in a program $\mathbb{P}$ by 'desaturation' of the saturated distribution tree for A in $\mathbb{P}$. The coalgebraic definition, for which we refer to [BZ15], applies pointwise on the saturated tree the counit $\epsilon_{\mathcal{U}\mathsf{At}} \colon \mathcal{U}\mathcal{K}\mathcal{U}\mathsf{At} \to \mathcal{U}\mathsf{At}$ of the adjunction $\mathcal{U} \dashv \mathcal{K}$ (*cf.* (4.3)). The TM distribution tree which results from 'desaturation' can be described very simply: at each layer of the starting saturated distribution tree, one prunes all the subtrees which are not labelled with the identity substitution $id := x_1 \mapsto x_1, x_2 \mapsto x_2, \dots$. In this way, the only remaining computation are those in which resolution only applies a non-trivial substitution on the clause side, that is, in which unification is restricted to term-matching.

Computability of term-matching distribution semantics. One may compute the success probability $\mathsf{Pr}^{\mathsf{TM}}_{\mathbb{P}}(A)$ in $\mathbb{P}$ from the TM distribution tree of A in $\mathbb{P}$. The computation goes similarly to Algorithm 2 : the problem amounts to calculating the probabilities of those deterministic subtrees of the distribution tree which prove the goal. We confine ourselves to some remarks on the aspects that require extra care, compared to the ground case.

(1) The probability $\mathsf{Pr}^{\mathsf{TM}}_{\mathbb{P}}(A)$ is not computable in whole generality. It depends on whether one can decide all the proofs of A in the pure logic program $|\mathbb{P}|$, and there are various heuristics in logic programming for this task.
(2) It is still possible to decide whether a subtree is deterministic, but the algorithm in the general case is a bit subtler, as it is now possible that two different goals match the same clause (instantiated in two different ways).
(3) When calculating the probability of a deterministic subtree in the TM distribution tree, multiple appearances of a single clause (possibly instantiated with different substitutions) should be counted only once. In order to ensure this one needs to be able to identify which clause is applied at each step of the computation described by the distribution trees: this is precisely the reason of the addition of the clause labels in the coalgebra type of these trees, as discussed in Section 4.3.

We conclude by briefly discussing the problem-solving perspective, in which resolution is based on arbitrary unification rather than just term-matching. In standard SLD-resolution, computability relies on the possibility of identifying the *most general* unifier between a goal

and the head of a given clause. This can be done also within saturated distribution trees, since saturation supplies *all* the unifiers, thus in particular the most general one. This means that, in principle, one may compute the distribution semantics based on most general unification from the saturated distributed tree associated with a goal, with similar caveats as the ones we described for the term-matching case. However, the lack of a satisfactory coalgebraic treatment of most general unifiers [BZ15] makes us prefer the theorem-proving perspective discussed above, for which desaturation provides an elegant categorical formalisation.

Appendix C. Computability of the weight semantics

In this appendix we will show how the weight $\omega(A)$ of a goal A is computable from its derivation semantics, i.e. the weighted derivation tree $[\![A]\!]_w$. We focus on the ground case, and then conclude with a discussion on how the generalisation to the variable case goes.

One preliminary consideration concerns cycles in WLP. Indeed, in a weighted derivation tree it may be the case that an atom-node has a directed path to another atom-node labelled with the same atom. While different ways to assign meaning to cycles may be justifiable (for instance in the context of (co)inductive logic programming), the standard choice (see e.g. [CSS10]) is to regard any proof subtree containing a cycle as failing to prove the given goal. In concrete, this means that proof subtrees with cycles may be simply discarded when calculating the weight of the goal.

To make this precise in our computation, recall that a proof subtree for a goal A is a subtree of $[\![A]\!]_w$ where we select exactly one child for each atom-node with children. We may define the weight of a proof subtree for A as the weight of A in that particular tree, calculated using (5.1). Note the overall weight of a goal A in $[\![A]\!]_w$ is then the $\oplus$-sum of the weights of its proof subtrees.

This means that, from a computational viewpoint, discarding a proof subtree is equivalent to assigning weight $\mathbf{0}$ (the $\oplus$-unit) to that proof subtree. In order to assign weight $\mathbf{0}$ to each proof subtree with cycles, we look for the first appearance of a cycle, re-label the next edge with $\mathbf{0}$, and discard all the descendants. This works because, according to (5.1) and the fact that $\mathbf{0} \otimes c = c \otimes \mathbf{0} = \mathbf{0}$ for any $c \in K$, any finite proof subtree containing an edge labelled with $\mathbf{0}$ has weight $\mathbf{0}$. We illustrate this idea with the following example.

Example C.1. Recall $\mathbb{P}^{gsp}$ in Example 5.4, and consider the goal $\mathsf{reachable}(\mathsf{c})$. Its weighted derivation tree $[\![\mathsf{reachable}(\mathsf{c})]\!]_{\mathbb{P}^{\mathsf{gsp}}}$ includes a finite path (subtree) witnessing a proof of the goal.

$$\mathsf{reach}(\mathsf{c}) \xrightarrow{4} \bullet \to \mathsf{reach}(\mathsf{a}) \xrightarrow{0} \bullet \to \mathsf{init}(\mathsf{a}) \xrightarrow{0} \bullet$$

Intuitively, this says that c is reachable from the initial state a in in (5.2), with weight 4. However, in $[\![\mathsf{reachable}(\mathsf{c})]\!]_{\mathbb{P}^{\mathsf{gsp}}}$ there are also other proof subtrees, which feature cycles, as for instance

$$\mathsf{reach}(\mathsf{c}) \xrightarrow{4} \bullet \to \mathsf{reach}(\mathsf{a}) \xrightarrow{9} \bullet \to \mathsf{reach}(\mathsf{c}) \xrightarrow{4} \bullet \to \mathsf{reach}(\mathsf{a}) \xrightarrow{0} \bullet \to \mathsf{init}(\mathsf{a}) \xrightarrow{0} \bullet \tag{C.1}$$

$$\mathsf{reach}(\mathsf{c}) \xrightarrow{4} \bullet \to \mathsf{reach}(\mathsf{a}) \xrightarrow{9} \bullet \to \mathsf{reach}(\mathsf{c}) \xrightarrow{4} \bullet \to \mathsf{reach}(\mathsf{a}) \xrightarrow{9} \bullet \to \mathsf{reach}(\mathsf{c}) \to \cdots \tag{C.2}$$

Our algorithm for computing the weight of $w(\mathsf{reachable}(\mathsf{c}))$ will prune both (C.1) and (C.2), letting them both become equal to

$$\mathsf{reach}(\mathsf{c}) \xrightarrow{4} \bullet \to \mathsf{reach}(\mathsf{a}) \xrightarrow{+\infty} \bullet \tag{C.3}$$

where $+\infty$ is the element $\mathbf{0}$ in the semiring of $\mathbb{P}^{gsp}$. To see that this procedure yields the correct result, note that the weight of the proof subtree (C.3) is $4 + (+\infty) = +\infty$, which does not affect the calculation of the weight $(= 4)$ of $\mathsf{reachable}(\mathsf{c})$.

Now we give the generic algorithms which compute the weight of a goal A given its weighted derivation tree $[\![A]\!]_w$ in the program $\mathbb{W}$. For readability, we divide our approach into 2 algorithms.

Algorithm 3 Prune a weighted derivation tree

Input: a weighted derivation tree $\mathcal{T}$
Output: $\mathcal{T}$ pruned all cycles

```
1: for path π in 𝒯 do
2:     atom_list = [ ]
3:     for node v in π do
4:         if label(v) in atom_list then
5:             t = grandparent of v, u = parent of v
6:             cut the descendants of u from 𝒯
7:             re-label the edge t → u by 0
8:         else
9:             atom_list.append(label(v))
10: return tree 𝒯
```

Algorithm 4 Calculate weight of the root node from a pruned weighted derivation tree

Input: the weighted derivation tree $\mathcal{T}$ for the goal A
Output: the weight of A

```
1: Prune 𝒯 using Algorithm 3
2: function Weight(x, 𝒯)
3:     sum_list = [ ]
4:     for children y of x do
5:         prod_list = [label(x → y)]            ▷ first get the weight of the clause
6:         for children z of y do
7:             prod_list.append(Weight(z, 𝒯))
8:             cur_weight = ⨂ prod_list          ▷ calculate the ⊗-product of all the values in prod_list
9:         sum_list.append(cur_weight)
10:    return ⨁ sum_list                         ▷ return the ⊕-sum of all the values in product_list
11: s = root of 𝒯                               ▷ calculate the weight of node s in 𝒯
12: return Weight(s, 𝒯)
```

Algorithm 3 uses depth-first search to find the first node whose atom label already appeared in some of its ancestors. This search procedure terminates because the set At of atoms is finite. Since a weighted derivation tree is finitely branching, the whole algorithm also terminates.

Algorithm 4 is a typical divide-and-conquer algorithm, where the weight of an atom-node is calculated via the weights of its (atom-)grandchildren. The procedure is simply spelling out (5.1) in the weighted derivation tree.

We conclude by sketching how this algorithm generalises to one for general WLP programs. First, recall the theorem-proving and problem-solving perspectives on logic programming, as discussed in Appendix B. For WLP, the theorem-proving aspect is prevalent in most applications (for example [CSS10, EB20, DEKM98]). Within this perspective, the task is to compute the weight of the goal itself, rather than the weight of some substitution instance of the goal. To compute such term-matching weight semantics, say for a goal A , one should first apply desaturation (as in Appendix B, see also [BZ15]) to the weighted saturated derivation tree $[\![A]\!]_{\mathsf{u}^\sharp}$, obtaining a new tree T_A. Concretely, T_A is obtained from $[\![A]\!]_{\mathsf{u}^\sharp}$ by pruning any subtree which is rooted by substitutions other than the identity substitution.

Second, one may tweak Algorithm 3 and 4 for ground WLP in order to deal with the substitution-nodes in 'desaturated' weighted derivation trees. The term-matching weight semantics of A can then be computed by applying the modified algorithms to T_A. Note that to guarantee termination of the computation, we require that for every atom A, there are only finitely many grounding $H\tau \leftarrow B_1\tau, \ldots, B_k\tau$ of the clauses whose heads $H\tau$ is A.

Appendix D. Missing proofs

Proposition D.1. *The operation* $\mathsf{pw} : \mathcal{M}_{pr} \Rightarrow \mathcal{D}_{\leq 1}\mathcal{P}_f$ *defined in Definition 3.12 is a natural transformation.*

Proof. The naturality for pw boils down to showing that for arbitrary $h\colon X \to Y$, $\mathcal{D}_{\leq 1}\mathcal{P}_f(h) \circ \mathsf{pw}_X = \mathsf{pw}_Y \circ \mathcal{M}_{pr}(h)\colon \mathcal{M}_{pr}(X) \to \mathcal{D}_{\leq 1}\mathcal{P}_f(Y)$. In this proof we fix an arbitrary function $h\colon X \to Y$. We start from an arbitrary $\varphi \in \mathcal{M}_{pr}(X)$. The case where $\mathsf{supp}(\varphi) = \emptyset$ is easy. So we assume that φ has a finite non-empty support $\mathsf{supp}(\varphi)$. On one hand, $\mathsf{pw}_X(\varphi)$ has support $\mathcal{P}(\mathsf{supp}(\varphi))$, and for any $A \in \mathcal{P}(\mathsf{supp}(\varphi))$, $\mathsf{pw}_X(\varphi)(A) = \big(\prod_{a\in A}\varphi(a)\big) \cdot \Big(\prod_{a\in\mathsf{supp}(\varphi)\setminus A}(1-\varphi(a))\Big)$. Then the support of $\mathcal{D}_{\leq 1}\mathcal{P}_f(h)(\mathsf{pw}_X(\varphi))$ is $h[\mathsf{supp}(\mathsf{pw}_X(\varphi))] = h[\mathcal{P}(\mathsf{supp}(\varphi))]$, and for each $B \in h[\mathcal{P}(\mathsf{supp}(\varphi))]$,

$$\begin{aligned}
\mathcal{D}_{\leq 1}\mathcal{P}_f(h)(\mathsf{pw}_X(\varphi))(B) &= \sum_{\substack{A\in\mathsf{supp}(\mathsf{pw}_X(\varphi))\\ h[A]=B}} \mathsf{pw}_X(\varphi)(A) \\
&= \sum_{\substack{A\in\mathcal{P}(\mathsf{supp}(\varphi))\\ h[A]=B}} \mathsf{pw}_X(\varphi)(A) \\
&= \sum_{\substack{A\in\mathcal{P}(\mathsf{supp}(\varphi))\\ h[A]=B}} \left(\prod_{a\in A}\varphi(a)\cdot \prod_{a\in\mathsf{supp}(\varphi)\setminus A}(1-\varphi(a))\right) \qquad \text{(D.1)}
\end{aligned}$$

On the other hand, $\mathcal{M}_{pr}(h)(\varphi)$ has support $h[\mathsf{supp}(\varphi)]$, and for arbitrary $y \in h[\mathsf{supp}(\varphi)]$, $\mathcal{M}_{pr}(h)(\varphi)(y) = \bigvee_{x\in h^{-1}(y)}\varphi(x) = 1 - \prod_{x\in h^{-1}(y)}(1-\varphi(x))$. Then $\mathsf{pw}_Y(\mathcal{M}_{pr}(h)(\varphi))$ has support $\mathcal{P}(\mathsf{supp}(\mathcal{M}_{pr}(h)(\varphi))) = \mathcal{P}(h[\mathsf{supp}(\varphi)])$. For each $B' \in \mathcal{P}(h[\mathsf{supp}(\varphi)])$,

$$\mathsf{pw}_Y(\mathcal{M}_{pr}(h)(\varphi))(B') = \prod_{y\in B'} \mathcal{M}_{pr}(h)(\varphi)(y) \cdot \prod_{y\in h[\mathsf{supp}(\varphi)]\setminus B'} (1-\mathcal{M}_{pr}(h)(\varphi)(y)) \tag{D.2}$$

Note that the two supports $h[\mathcal{P}(\mathsf{supp}(\varphi))]$ and $\mathcal{P}(h[\mathsf{supp}(\varphi)])$ are equivalent: $B \in h[\mathcal{P}(\mathsf{dom}(\varphi))]$ if and only if $\exists A \subseteq \mathsf{dom}(\varphi)$ such that $h[A] = B$; $B' \in \mathcal{P}(h[\mathsf{supp}(\varphi)])$ if and only if $B' \subseteq h[\mathsf{supp}(\varphi)]$ if and only if $\exists A' \subseteq \mathsf{supp}(\varphi)$ such that $h[A'] = B'$.

We spell out the operation of $\mathcal{M}_{pr}$ on h in (D.2), and the steps (D.3), (D.4) below will be explained in detail after the calculation:

$$\begin{aligned}
&\mathsf{pw}_Y(\mathcal{M}_{pr}(h)(\varphi))(B') \\
&= \prod_{y\in B'} \left(1 - \prod_{x\in h^{-1}(y)} (1-\varphi(x))\right) \cdot \prod_{y\in h[\mathsf{supp}(\varphi)]\setminus B'} \left(1 - \left(1 - \prod_{x\in h^{-1}(y)} (1-\varphi(x))\right)\right) \\
&= \prod_{y\in B'} \left(1 - \prod_{x\in h^{-1}(y)} (1-\varphi(x))\right) \cdot \prod_{y\in h[\mathsf{supp}(\varphi)]\setminus B'} \prod_{x\in h^{-1}(y)} (1-\varphi(x)) \\
&\overset{(D.5)}{=} \prod_{y\in B'} \left(\sum_{Z\subseteq_i h^{-1}(y)} \left(\prod_{x\in Z} \varphi(x) \cdot \prod_{x\in h^{-1}(y)\setminus Z} (1-\varphi(x))\right)\right) \cdot \prod_{\substack{y\in h[\mathsf{supp}(\varphi)]\setminus B' \\ x\in h^{-1}(y)}} (1-\varphi(x)) \qquad \text{(D.3)} \\
&= \sum_{\substack{A'\subseteq \mathsf{supp}(\varphi) \\ h[A']=B'}} \left(\prod_{x\in A'} \varphi(x) \cdot \prod_{x\in h^{-1}[B]\setminus A'} (1-\varphi(x))\right) \cdot \prod_{\substack{y\in h[\mathsf{supp}(\varphi)]\setminus B' \\ x\in h^{-1}(y)}} (1-\varphi(x)) \qquad \text{(D.4)} \\
&= \sum_{\substack{A'\subseteq \mathsf{supp}(\varphi) \\ h[A']=B'}} \left(\prod_{x\in A'} \varphi(x) \cdot \prod_{x\in h^{-1}[B']\setminus A'} (1-\varphi(x))\right) \cdot \prod_{\substack{x\in \mathsf{supp}(\varphi) \\ x\notin h^{-1}[B']}} (1-\varphi(x)) \\
&= \sum_{\substack{A'\subseteq \mathsf{supp}(\varphi) \\ h[A']=B'}} \left(\prod_{x\in A'} \varphi(x) \cdot \prod_{x\in h^{-1}[B']\setminus A'} (1-\varphi(x)) \cdot \prod_{\substack{x\in \mathsf{supp}(\varphi) \\ x\notin h^{-1}[B']}} (1-\varphi(x))\right) \\
&= \sum_{\substack{A'\subseteq \mathsf{supp}(\varphi) \\ h[A']=B'}} \left(\prod_{x\in A'} \varphi(x) \cdot \prod_{x\in \mathsf{supp}(\varphi)\setminus A'} (1-\varphi(x))\right) \\
&\overset{(D.1)}{=} \mathcal{D}_{\leq 1}\mathcal{P}_f(h)(\mathcal{M}_{pr}(\varphi))(B')
\end{aligned}$$

In (D.3), $\subseteq_i$ is the relation $\subseteq$ restricted to non-empty subsets. Formally, (D.3) is obtained via the following equation, for arbitrary set X, $\psi \in \mathcal{M}_{pr}(X)$ and set $A \subseteq \mathsf{supp}(\psi)$:

$$1 - \prod_{a\in A} (1-\psi(a)) = \sum_{Z\subseteq_i A} \left(\prod_{a\in Z} \psi(a) \cdot \prod_{a\in A\setminus Z} (1-\psi(a))\right) \tag{D.5}$$

Intuitively, one can think of A as a set of independent events, and $\psi(a)$ as the probability of the event $a \in A$. Then the left-hand-side of (D.5) calculates 1 minus the probability that none of $a \in A$ happens; the right-hand-side sums up the probabilities of all possible worlds in which some of $a \in A$ happens. Both calculate the same value, namely the probability that at least one $a \in A$ happens.

The expression (D.4) is obtained by expanding $\prod_{y\in B'} \sum_{Z\subseteq_i h^{-1}(y)} G(y,Z)$ into a summation, where $G(y,Z) = \prod_{x\in Z} \varphi(x) \cdot \prod_{x\in h^{-1}(y)\setminus Z}(1-\varphi(x))$. In the resulting summation, each summand chooses exactly one element from each $\sum_{Z\subseteq_i h^{-1}(y)} G(y,Z)$, for all $y \in B'$. Moreover, every such choice corresponds to exactly one A' satisfying $h[A'] = B'$. So we have

$$\begin{aligned}\prod_{y\in B'} \sum_{Z\subseteq_i h^{-1}(y)} G(y,Z) &= \sum_{\substack{A'\subseteq \mathsf{supp}(\varphi)\\ h[A']=B'}} \prod_{y\in B'} G(y, h^{-1}(y)\cap A') \\ &= \sum_{\substack{A'\subseteq \mathsf{supp}(\varphi)\\ h[A']=B'}} \left(\prod_{y\in B'} \left(\prod_{x\in h^{-1}(y)\cap A'} \varphi(x) \cdot \prod_{x\in h^{-1}(y)\setminus(h^{-1}(y)\cap A')} (1-\varphi(x))\right)\right) \\ &= \sum_{\substack{A'\subseteq \mathsf{supp}(\varphi)\\ h[A']=B'}} \left(\prod_{x\in A'} \varphi(x) \cdot \prod_{x\in h^{-1}[B']\setminus A'} (1-\varphi(x))\right)\end{aligned}$$

and this is exactly the summation in (D.4).

Therefore $(\mathcal{D}_{\leq 1}\mathcal{P}_f(h) \circ \mathsf{pw}_X)(\varphi)$ and $(\mathsf{pw}_Y \circ \mathcal{M}_{pr}(h))(\varphi)$ have the same support, and both have the same values for arbitrary B in the support. □

This work is licensed under the Creative Commons Attribution License. To view a copy of this license, visit https://creativecommons.org/licenses/by/4.0/ or send a letter to Creative Commons, 171 Second St, Suite 300, San Francisco, CA 94105, USA, or Eisenacher Strasse 2, 10777 Berlin, Germany