Functorial Semantics as a Unifying Perspective on Logic Programming

Tao Gu
University College London, UK

Fabio Zanasi
University College London, UK

Abstract

Logic programming and its variations are widely used for formal reasoning in various areas of Computer Science, most notably Artificial Intelligence. In this paper we develop a systematic and unifying perspective for (ground) classical, probabilistic, weighted logic programs, based on categorical algebra. Our departure point is a formal distinction between the syntax and the semantics of programs, now regarded as separate categories. Then, we are able to characterise the various variants of logic program as different *models* for the same syntax category, i.e. structure-preserving functors in the spirit of Lawvere's functorial semantics.

As a first consequence of our approach, we showcase a series of semantic constructs for logic programming pictorially as certain string diagrams in the syntax category. Secondly, we describe the correspondence between probabilistic logic programs and Bayesian networks in terms of the associated models. Our analysis reveals that the correspondence can be phrased in purely syntactical terms, without resorting to the probabilistic domain of interpretation.

2012 ACM Subject Classification

Keywords and phrases string diagrams, functorial semantics, logic programming

Digital Object Identifier 10.4230/LIPIcs.CALCO.2021.

1 Introduction

Logic programming is a programming paradigm widely used for knowledge representation in Artificial Intelligence and related fields. In 'classical' logic programming, at the basis of formalisms such as Prolog [8] and Datalog [9], programs are sets of Horn clauses, of the form $A \leftarrow B_1, B_2, \ldots, B_n$. However, in the last two decades, various extensions and variants of logic programming emerged in order to handle non-classical reasoning, in which clauses may be associated with a probability or a weight, resulting in a different semantics. These approaches include probabilistic logic programming ([35], Problog [13], PRISM [36], CP-logic [31], PASP [12]) and weighted logic programming [15]. Seemingly different formalisms, such as Bayesian networks, also turn out to be closely related [31].

The main goal of this work is to develop a systematic, unifying framework for the different families of logic programming languages, in which their similarities and differences can be analysed algebraically using the abstract perspective of *category theory*.

Our approach is based on a simple, yet fruitful insight: a formal distinction between the *syntax* and the *semantics* of logic programs. This 'separation of concerns' has the benefit of clearly isolating the purely inferential structure underlying a program (syntax) from its 'type' (classical, probabilistic, or weighted, expressed by the semantics). Our guiding principle is the perspective of *functorial semantics*, as pioneered by Lawvere [29], in which one encodes an algebraic theory as a category Syn of 'syntactic' morphisms (tuples of terms), and then studies models of the theory as *functors* from Syn — the requirement that such functors preserve finite products makes them adhere to the usual notion of model in universal algebra. In analogy, from a logic program $\mathbb{P}$ we will freely generate a category SynLP of *string diagrams* [37], capturing the inferential structure of $\mathbb{P}$ as (graphical) syntax. Then, we consider models of SynLP, i.e. structure-preserving functors into other categories, acting as semantic domains of interpretation.

Our main conceptual contribution is the realisation that the different flavours of logic program (classical, probabilistic and weighted) amount to different classes of models of the same syntax category.

This perspective has various consequences. As the syntax is expressed as a freely generated category of string diagrams, not only it includes the clauses of the program as morphisms, but it allows to combine them into more elaborated representations. This provides much flexibility in expressing various kinds of semantic constructs, which may be observed in the image of the models/functors. We use this observation to provide a string diagrammatic description of semantics commonly found in the logic programming literature, such as the immediate consequence operator, the Herbrand semantics, and the stratified semantics, see e.g. [17, 3, 18, 38], as well as the distribution semantics of probabilistic programs, and the standard semantics of weighted programs.

Another payoff of our approach is providing an original perspective on the correspondence between probabilistic logic programs and Bayesian networks. It is a folklore result (*cf.* [31]) that the two formalisms can be translated one into another, modulo some caveats. In the context of our framework, we take advantage of the fact that Bayesian networks are also susceptible of a description in terms of functorial semantics, see [16, 22], and study their correspondence with probabilistic programs in terms of the associated models. Again, the distinction between syntax and semantics provides a valuable insight: it turns out that the correspondence can be entirely expressed at the level of the syntax categories — in contrast with traditional approaches, where it involves a mixture of transformations between graphs, programs, and conditional probabilities. Furthermore, thanks to the use of *string diagrams*, both the combinatorial structure of Bayesian networks (directed acyclic graphs) and the syntax of probabilistic programs can be expressed uniformly as entities of the same kind.

Synopsis. Section 2 is for preliminaries. We introduce the syntax category and present the functorial semantics of classical logic programming in Section 3. We then consider probabilistic logic programming in Section 4, and study the correspondence with Bayesian networks at a functorial level. Section 5 briefly illustrates the case of weighted logic programming. Section 6 is devoted to future work. Missing proofs may be found in Appendix B.

2 Preliminaries

Logic programming. We briefly recall the basics of logic programming, and refer to [33] for more details. Throughout this paper we focus on *ground* logic programming, i.e. in which programs have no variables. A *logic program* (LP) $\mathbb{P}$ based on a set of atoms At is a finite set of clauses φ of the form $A \leftarrow L_1, \dots, L_m.$, where A is an atom and each L_i is a literal (an atom B or a negated atom $\neg B$). The atom A and the set of literals $\{L_1, \dots, L_m\}$ are called the head (denoted as $\mathsf{head}(\varphi)$) and the body (denoted as $\mathsf{body}(\varphi)$) of the clause, respectively. A clause is *definite* if all the literals in its body are positive (namely atoms), and $\mathbb{P}$ is a *definite* if all its clauses are definite. We write $\mathbb{P}' \subseteq \mathbb{P}$ to mean that $\mathbb{P}'$ is a sub-program of $\mathbb{P}$, and $\mathcal{P}(\mathbb{P})$ for the set of all sub-programs of $\mathbb{P}$.

An *interpretation* $\mathcal{I}$ is a subset of At, and it is a *model* of $\mathbb{P}$ if for all $\varphi \in \mathbb{P}$, $\mathsf{body}(\varphi) \subseteq \mathcal{I}$ implies $\mathsf{head}(\varphi) \in \mathcal{I}$. A literal L is true in $\mathcal{I}$, denoted as $\mathcal{I} \vDash L$, if either $L = B$ and $B \in \mathcal{I}$, or $L = \neg B$ and $B \notin \mathcal{I}$, for some $B \in At$. Suppose X is a set of literals, we define $\mathcal{I} \vDash X$ if $\mathcal{I} \vDash L$ for all $L \in X$. There is an inclusion ordering on models, and every definite logic program $\mathbb{P}$ has a least model (referred to its least Herbrand semantics), denoted as $\mathcal{H}(\mathbb{P})$. Least models of definite logic programs can also be characterised as the least fixed points of *immediate consequence operators.* The immediate consequence operator $\mathbf{T}_{\mathbb{P}}$ associated to a logic program $\mathbb{P}$ with atom set At is a function $\mathcal{P}(At) \to \mathcal{P}(At)$ such that for any interpretation $\mathcal{I}$, $\mathbf{T}_{\mathbb{P}}(\mathcal{I}) := \{A \mid \exists \psi \in \mathbb{P} \text{ such that } \mathsf{head}(\psi) = A \text{ and } \mathcal{I} \vDash \mathsf{body}(\psi)\}$.

Note that arbitrary, non-definite logic programs may not have a least model. There are various alternative denotational semantics, including stratified semantics [3, 30], stable semantics [18], well-founded semantics [38]. We recall the stratified semantics from [3]. Given a LP $\mathbb{P}$, the *definition* of an atom A is the sub-program $def(A) := \{\varphi \in \mathbb{P} \mid \mathsf{head}(\varphi) = A\}$. $\mathbb{P}$ is *stratified* if there exists a partition $At_1, \ldots, At_k$ (called a stratification) of At such that,

- If there exists $\mathbb{P}_i$-clause φ such that $\neg B \in \mathsf{body}(\varphi)$, then $def(B) \subseteq \bigcup_{j<i} At_j$.
- If there exists $\mathbb{P}_i$-clause φ such that $B \in \mathsf{body}(\varphi)$, then $def(B) \subseteq \bigcup_{j\leq i} At_j$.

where $\mathbb{P}_1, \ldots, \mathbb{P}_k$ is the partition of $\mathbb{P}$ induced by the stratification: each $\mathbb{P}_i$ is $\{\varphi \in \mathbb{P} \mid \mathsf{head}(\varphi) \in At_i\}$. Then the *stratified model* $\mathcal{S}(\mathbb{P})$ of $\mathbb{P}$ is defined as $\mathcal{S}(\mathbb{P}) = \cup_{i=1}^{k} M_i$, where:

- $M_1 := \mathcal{H}(\mathbb{P}_1)$, and it is well-defined because $\mathbb{P}_1$ is always definite.
- $M_{i+1} := \mathcal{H}(\mathbb{Q}_{i+1})$, where $\mathbb{Q}_{i+1}$ is obtained by (1) deleting all the $\mathbb{P}_{i+1}$ clauses whose bodies contain some literals false in $\cup_{j=1}^{i} M_j$, and (2) in the remaining clauses, delete all the literals not in $\mathbb{P}_{i+1}$.

Importantly, the stratified semantics is independent of the choice of a specific stratification.

Probabilistic logic programming. A *probabilistic logic program* (PLP) $\mathbb{P}$ is a set of probabilistic clauses ψ of the form $p :: \varphi$, where $p \in [0,1]$ is a real number, and φ is a clause. p is called the (probabilistic) label of clause ψ, denoted as $\mathsf{Lab}(\psi)$. By forgetting the labels in $\mathbb{P}$, we obtain the pure logic program $|\mathbb{P}|$ of $\mathbb{P}$.

We will use the 'bra-ket' (or Dirac) notation for probability distributions, e.g. $0.7|a\rangle + 0.3|b\rangle$ is the distribution on $\{a, b\}$ assigning probability 0.7 to a and 0.3 to b. We recall the distribution semantics. A PLP $\mathbb{P}$ determines a probability distribution $\mu_{\mathbb{P}}$ over the sub-programs of $|\mathbb{P}|$: for any $\mathbb{L} \subseteq |\mathbb{P}|$, $\mu_{\mathbb{P}}(\mathbb{L}) := \left(\prod_{\varphi\in\mathbb{L}} \mathsf{Lab}(\varphi)\right) \cdot \left(\prod_{\varphi\in|\mathbb{P}|\setminus\mathbb{L}} (1 - \mathsf{Lab}(\varphi))\right)$. If $\mathbb{P}$ is acyclic, $\mu_{\mathbb{P}}$ allows to compute the probability that a given goal is proved. The *success probability* $\pi_{\mathbb{P}}(\bar{L})$ (or simply probability) of a goal $\bar{L} = L_1 \wedge \cdots \wedge L_m$ is $\sum \{\mu_{\mathbb{P}}(\mathbb{L}) \mid \mathbb{L} \subseteq |\mathbb{P}|, \mathcal{S}(\mathbb{L}) \vDash L_1, \ldots, L_m\}$. The distribution $\delta_{\mathbb{P}}(\bar{A})$ of a set of atoms $\bar{A} = \{A_1, \ldots, A_m\}$ over its interpretations is thus defined as $\sum_{L_i \in \{A_i, \neg A_i\}} \pi_{\mathbb{P}}(L_1 \wedge \cdots \wedge L_m)|L_1, \ldots, L_m\rangle$.

▶ **Example 1.** The PLP program $\mathbb{P}_{\text{wet}}$ below describes how the season affects the probability of raining and a sprinkler to leak, which cause the grass to be wet and the road to be slippery.

0.25 ::	`Winter`	$\leftarrow$.	(ψ_1)	0.9 ::	`WetGrass`	$\leftarrow$ `Sprinkler`.	(ψ_5)
0.2 ::	`Sprinkler`	$\leftarrow$ `Winter`.	(ψ_2)	0.8 ::	`WetGrass`	$\leftarrow$ `Rain`.	(ψ_6)
0.6 ::	`Rain`	$\leftarrow$ `Winter`.	(ψ_3)	0.7 ::	`SlipperyRoad`	$\leftarrow$ `Rain`.	(ψ_7)
0.1 ::	`Rain`	$\leftarrow \neg$`Winter`.	(ψ_4)	0.1 ::	`SlipperyRoad`	$\leftarrow \neg$`Rain`.	(ψ_8)

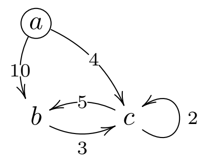
If we forget about all the probabilistic labels, then we get a logic program $|\mathbb{P}_{\text{wet}}|$. For instance, we can calculate the success probability $\pi_{\mathbb{P}_{\text{wet}}}(\texttt{Winter} \wedge \texttt{WetGrass}) = 0.1434$.

Weighted Logic Programming. A weighted logic program (WLP) $\mathbb{P}$ based on an ω-complete commutative semiring $\mathcal{K} = \langle K, +, \cdot, \mathbf{0}, \mathbf{1}\rangle$ is a finite set of weighted clauses φ of the form $w :: A \leftarrow B_1, \cdots, B_k.$, where $w \in K$ is the weight label of the clause (denoted as $lab(\varphi)$). We also assume that $\mathcal{K}$ is a complete lattice under the ordering "$x \preceq y$ if $x + y$". Given atoms $At = \{A_1, \ldots, A_n\}$, for each A_i, let $\mathcal{K}_{A_i}$ be a copy of $\mathcal{K}$ standing for the values of atom A_i, and we write $\mathcal{K}_{A_i}$ as $\{w_{A_i} \mid w \in \mathcal{K}\}$. A *weight state* is a tuple $u \in \mathcal{K}_{A_1} \times \cdots \times \mathcal{K}_{A_n}$, and its i-th component is referred to as $u(A_i)$.

The semantics of WLP assigns weights to atoms that are defined as the least fixed point of a weighted variant $\mathbf{T}^w$ of the immediate consequence operator for classical logic programs [15]. Given a WLP program $\mathbb{P}$ with atom set $\{A_1, \ldots, A_n\}$, $\mathbf{T}^w_{\mathbb{P}}$ is a function $\mathcal{K}_{A_1} \times \cdots \times \mathcal{K}_{A_n} \to \mathcal{K}_{A_1} \times \cdots \times \mathcal{K}_{A_k}$ such that for each weight state u, the i-th component of $\mathbf{T}^w_{\mathbb{P}}(u)$ is $\sum\{lab(\varphi) \cdot u(B_1) \cdot \cdots \cdot u(B_k) \mid \mathsf{head}(\varphi) = A_i, \mathsf{body}(\varphi) = \{B_1, \ldots, B_k\}\}$. The *weight* $weight_{\mathbb{P}}(A_i)$ of A_i is then the i-th component of the least fixed point of $\mathbf{T}^w_{\mathbb{P}}$, which exists because $\mathbf{T}^w_{\mathbb{P}}$ is a monotonic function on a complete lattice $\mathcal{K}_{A_1} \times \cdots \times \mathcal{K}_{A_n}$.

▶ **Example 2.** The semiring $\mathcal{K}^{sp} = \langle \mathbb{N} \cup \{+\infty\}, \min, +, +\infty, 0\rangle$ is a fragment of the 'min-plus' tropical semiring, used for shortest path problems. Consider the WLP program $\mathbb{P}_{sp}$ consisting of all *the grounding of* the clauses below left in (1), which describes the reachability condition of the weighted directed graph D below right in (1). Then the answer to the shortest path from the initial state to a state `x` can be calculated by the weight of `reachable(x)` in $\mathbb{P}_{sp}$. For instance, $weight_{\mathbb{P}_{sp}}(\texttt{reachable(b)}) = 9$, which is the least path weight from `a` to `b` in D.

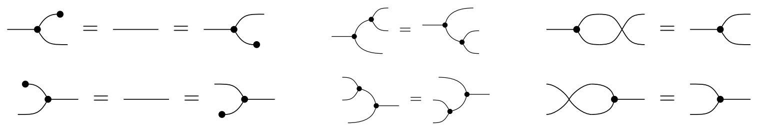
$$\begin{array}{rrl|rrl} 0 :: & \texttt{initial(a)} & \leftarrow . & 10 :: & \texttt{edge(a,b)} & \leftarrow . \\ 4 :: & \texttt{edge(a,c)} & \leftarrow . & 3 :: & \texttt{edge(b,c)} & \leftarrow . \\ 5 :: & \texttt{edge(c,b)} & \leftarrow . & 2 :: & \texttt{edge(c,c)} & \leftarrow . \\ 0 :: & \texttt{reachable(x)} & \leftarrow & \multicolumn{3}{l}{\texttt{initial(x)}.} \\ 0 :: & \texttt{reachable(x)} & \leftarrow & \multicolumn{3}{l}{\texttt{reachable(y)}, \texttt{edge(y,x)}.} \end{array}$$



(1)

Bayesian networks We will explore the relationship between PLP and Bayesian networks (BN), which we briefly recall. A Bayesian network $\mathbb{B}$ on a set $A_1, A_2, \ldots A_k$ of variables (finite sets) is a pair $(G, \Pr)$. Here G is a directed acyclic graph (DAG) (V_G, E_G), where $V_G = A_1, A_2, \ldots A_k$. The second component Pr is a family $\{\Pr(A \mid pa(A))\}_{A \in V_G}$ of conditional probability distributions, where $pa(A)$ is the set of predecessors of A according to E_G, and the distribution $\Pr(A \mid pa(A))$ assigns a probability to each element in the set A. We say that BN is *boolean-valued* when each variable A is a two-element set, which we write $\{A, \neg A\}$ with slight abuse of notation (meaning 'A is true', 'A is false').

CDMU and CD categories. We will formulate both the syntax and the semantics of logic program in terms of categories, equipped with the structure of a *CDMU category*. A CDMU category (for **c**opy, **d**iscard, **m**ultiplication, **u**nit) is a symmetric monoidal category $\langle \mathbf{C}, \otimes, I\rangle$ where each object C has a copier $-\!\!\bullet\!\!<_C : C \to C \otimes C$, a discarder $-\!\!\bullet_C : C \to I$, a multiplication $>\!\!\bullet\!\!-_C : C \otimes C \to C$, and a unit $\bullet\!\!-_C : I \to C$ — which we will often write omitting C, when clear from the context. These operations are required to satisfy the following set of equations:



(2)

In words, there are a commutative comonoid $(-\!\!\bullet\!\!<, -\!\!\bullet)$ and a commutative monoid $(>\!\!\bullet\!\!-, \bullet\!\!-)$. A *CDMU functor* between two CDMU categories is a symmetric monoidal functor that also preserves the commutative comonoid and monoid structures. We will often use the construction of the *free* CDMU category $\mathsf{freeCDMU}(X, \Sigma)$ over a generating set X of objects and a generating set Σ of morphisms: the objects are finite lists $[x_1, \ldots, x_n]$ over X (including the empty list $[\,]$), and the morphisms are string diagrams [37] freely obtained by composing morphisms in Σ together with $-\!\!\bullet\!\!<, -\!\!\bullet, >\!\!\bullet\!\!-, \bullet\!\!-$, modulo the equations in (2).

When describing Bayesian networks, we will also need to refer to *CD categories*. These are defined in the same way as CDMU categories, but without the multiplications $>\!\!\bullet\!\!-_C : C \otimes C \to C$ and units $\bullet\!\!-_C : I \to C$. We write $\mathsf{freeCD}(X, \Sigma)$ for the free CD category over objects X and morphisms Σ. See [22] and Appendix A for background on how CD categories relate to BNs.

3 Classical Logic Programming

In this section we introduce a *functorial semantics* of classical logic programs, which culminates in Proposition 5 below. Our starting point is a conceptual distinction between the syntax and the semantics of a logic program. In Subsection 3.1 we introduce the syntax category of string diagrams for a given logic program $\mathbb{P}$, which will also be used in Sections 4 and 5 for the probabilistic and the weighted case. Next, in Subsection 3.2 we provide a semantics category, and identifies logic programs with certain *models* — structure-preserving functors from the syntax category to the semantics category. Finally, in Subsection 3.3 we describe some well-known semantics (immediate consequence operator, least Herbrand semantics, stratified semantics) of logic programs as images of specific string diagrams in the syntax, under the functorial interpretation. This perspective will provide an original, diagrammatic representation for such semantic constructs.

3.1 Syntax Category

Every logic program $\mathbb{P}$ has an underlying definite logic program $[\mathbb{P}]$ describing the purely inferential structure of $\mathbb{P}$ — where, intuitively, we disregard information about probabilities, weights, and negated clauses. $[\mathbb{P}]$ is what is commonly used in the definition of the *dependency graph* of $\mathbb{P}$ [3], which describes the dependency relation between atoms in $\mathbb{P}$, and plays a key role in the definition of stratified logic programs [3]. To incorporate this step in our approach, first we define a 'forgetful' function τ^{gen} (resp. τ^{pr}, τ^{wt}) from arbitrary (classical, probabilistic, or weighted) clauses to definite clauses as follows:

$$\begin{array}{lcrll}
\tau^{gen} & : & A & \leftarrow B_1,...,B_k,\neg C_1,...,\neg C_\ell. & \mapsto \quad A \leftarrow B_1,...,B_k,C_1,...,C_\ell. \\
\tau^{pr} & : & p :: A & \leftarrow B_1,...,B_k,\neg C_1,...,\neg C_\ell. & \mapsto \quad A \leftarrow B_1,...,B_k,C_1,...,C_\ell. \\
\tau^{wt} & : & w :: A & \leftarrow B_1,...,B_k. & \mapsto \quad A \leftarrow B_1,...,B_k.
\end{array}$$

In words, τ forgets the quantitative labels and the negation. The definite logic program $[\mathbb{P}]$ is defined as the image of $\mathbb{P}$ under the appropriate τ, namely $[\mathbb{P}] := \{\tau(\varphi) \mid \varphi \in \mathbb{P}\}$. We say $\mathbb{P}$ is based on $[\mathbb{P}]$, or $[\mathbb{P}]$ is the underlying program of $\mathbb{P}$. Note that, if we read $A \leftarrow B_1, \dots, B_k.$ as 'nodes $B_1, \dots, B_k$ are parents of A', $[\mathbb{P}]$ defines exactly (the components of) the dependency graph of $\mathbb{P}$ — *cf.* [3]. Observe that there may exist distinct clauses in $\mathbb{P}$ that have the same image under τ, so the size of $[\mathbb{P}]$ is less or equal to that of $\mathbb{P}$.

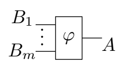
▶ **Example 3.** Recall $\mathbb{P}_{\text{wet}}$ from Example 1. The underlying definite logic program $[\mathbb{P}_{\text{wet}}]$ is

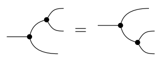
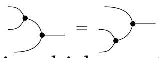
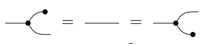
$$\begin{array}{lll@{\qquad}lll}
\texttt{Winter} & \leftarrow . & (\varphi_1) & \texttt{WetGrass} & \leftarrow \texttt{Sprinkler.} & (\varphi_4) \\
\texttt{Sprinkler} & \leftarrow \texttt{Winter.} & (\varphi_2) & \texttt{WetGrass} & \leftarrow \texttt{Rain.} & (\varphi_5) \\
\texttt{Rain} & \leftarrow \texttt{Winter.} & (\varphi_3) & \texttt{SlipperyRoad} & \leftarrow \texttt{Rain.} & (\varphi_6)
\end{array}$$

For instance, $\tau^{pr}(\psi_3) = \tau^{pr}(\psi_4) = (\texttt{Rain} \leftarrow \texttt{Winter.}) = \varphi_3$.

Given a definite logic program $\mathbb{L}$ on At, we construct a CDMU category $\mathsf{SynLP}_{\mathbb{L}}$ which encodes the inferential structure represented by $\mathbb{L}$. We define $\mathsf{SynLP}_{\mathbb{L}}$ as the free CDMU category $\mathsf{freeCDMU}(At, \Sigma_{\mathbb{L}})$ (*cf.* *Section 2*), where the set $\Sigma_{\mathbb{L}}$ of generating morphisms consists of one string diagram for each clause in $\mathbb{L}$:

$$\Sigma_{\mathbb{L}} := \left\{ \quad \middle| \; \varphi \equiv A \leftarrow B_1, \dots, B_m. \text{ is a clause in } \mathbb{L} \right\} \qquad (3)$$



With some abuse of notation, we use φ to refer to both a clause and its corresponding string diagram in $\Sigma_{\mathbb{L}}$. We choose to work with CDMU categories because their equations subsume structure that is always present in logic programs. For example,  and  reflect the intuition that there is no ordering on the (possibly multiple) clauses in which an atom may appear;  says that generating two occurrences of the same atom and then disregarding one is the same as just working with a single occurrence.

$\mathsf{SynLP}_{\mathbb{L}}$ will act as the syntax category for all $\mathbb{P}$ such that $[\mathbb{P}] = \mathbb{L}$. We will identify logic programs based on $\mathbb{L}$ with functors from $\mathsf{SynLP}_{\mathbb{L}}$ to some appropriate 'semantics categories', which will vary depending on whether the program is classical, probabilistic, or weighted.

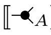
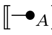
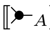
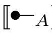
3.2 Functorial Semantics of LP

In this subsection we introduce a categorical semantics for classical logic programming, and characterise logic programs as functors to this semantics.

For the classical case, the semantics domain will the category $\mathbf{Set}(\mathbf{2})$ defined as follows. Objects of $\mathbf{Set}(\mathbf{2})$ are finite products $\mathbf{2}_{A_1} \times \cdots \times \mathbf{2}_{A_n}$, where each A_i is a two-elements Boolean algebra, which we write $\{A_i, \neg A_i\}$, with $A_i > \neg A_i$, to emphasise that the two elements will be treated as an atom and its negation. In particular, the singleton set $\mathbf{1} = \{*\}$ is the 0-ary product. Morphisms in $\mathbf{Set}(\mathbf{2})$ are simply functions between the underlying sets.

We write $\vee$, $\wedge$, $(\cdot)^-$ for the standard Boolean algebra operations. Note every $\mathbf{Set}(\mathbf{2})$-object is itself a finite Boolean algebra, with the operations defined pointwise.

Given a classical logic program $\mathbb{P}$ with underlying definite program $\mathbb{L} := [\mathbb{P}]$, we may associate $\mathbb{P}$ with a functor $[\![-]\!]_{\mathbb{P}} : \mathsf{SynLP}_{\mathbb{L}} \to \mathbf{Set}(\mathbf{2})$ defined as follows. On objects, $[\![-]\!]_{\mathbb{P}}$ maps $A \in At$ to $\mathbf{2}_A = \{A, \neg A\}$. On morphisms, the CDMU structure is interpreted as

$[\![$  $_A]\!]_{\mathbb{P}} : \mathbf{2}_A \to \mathbf{2}_A \times \mathbf{2}_A$, $x \mapsto (x, x)$ $[\![$  $_A]\!]_{\mathbb{P}} : \mathbf{2}_A \to \mathbf{1}$, $x \mapsto *$ $[\![$  $_A]\!]_{\mathbb{P}} : \mathbf{2}_A \times \mathbf{2}_A \to \mathbf{2}_A$, $(x, y) \mapsto x \vee y$ $[\![$  $_A]\!]_{\mathbb{P}} : \mathbf{1} \to \mathbf{2}_A$, $* \mapsto \neg A.$

For each $\mathbb{L}$-clause $\varphi \equiv A \leftarrow B_1, \ldots, B_m.$, $[\![\varphi]\!]_{\mathbb{P}}$ maps $u \in \mathbf{2}_{B_1} \times \cdots \times \mathbf{2}_{B_m}$ to A if u as an interpretation (*cf.* Sec. 2) satisfies $u \vDash \mathsf{body}(\psi)$ for some ψ with $\tau^{gen}(\psi) = \varphi$, and to $\neg A$ otherwise.

Note that, in order for $[\![-]\!]_{\mathbb{P}}$ to be well-defined, the semantic domain $\mathbf{Set}(\mathbf{2})$ should also satisfy the CDMU equations — see Lemma 27 in Appendix B for a proof.

▶ **Example 4.** The clause $\psi \equiv A \leftarrow B_1, \neg B_2.$ is associated with the definite clause $\varphi \equiv A \leftarrow B_1, B_2.$, and it gets interpreted as a function $[\![\varphi]\!] : \mathbf{2}_{B_1} \times \mathbf{2}_{B_2} \to \mathbf{2}_A$ mapping $(B_1, \neg B_2)$ to A and (B_1, B_2) to $\neg A$. Incidentally, note this is *not* a boolean function, as for instance $(\neg B_1, B_2) \vee (B_1, B_2) = (B_1, B_2)$ but $[\![\varphi]\!]\,(\neg B_1, B_2) \vee [\![\varphi]\!]\,(B_1, B_2) \neq [\![\varphi]\!]\,(B_1, B_2)$.

The next proposition formally states the correspondence characterising the functorial semantics of LP. In there, we call *generator-preserving* a functor $\mathbf{C} \to \mathbf{D}$ that maps generating objects of $\mathbf{C}$ to generating objects of $\mathbf{D}$ (assuming objects of the two categories are freely obtained from a set of generators). In our case, $\mathbf{C} = \mathsf{SynLP}_{\mathbb{L}}$, $\mathbf{D} = \mathbf{Set}(\mathbf{2})$ and the requirement ensures that a functor maps each atom $A \in At$, seen as object of $\mathsf{SynLP}_{\mathbb{L}}$, to a two-element Boolean algebra $\mathbf{2}$, which we write $\mathbf{2}_A = \{A, \neg A\}$ to emphasise this correspondence.

▶ **Proposition 5.** *There is a 1-1 correspondence between logic programs based on* $\mathbb{L}$ *and generator-preserving CDMU functors of type* $\mathsf{SynLP}_{\mathbb{L}} \to \mathbf{Set}(\mathbf{2})$.

Echoing the terminology of Lawvere's functorial semantics [29], we will refer to the functors as in Proposition 5 simply as *models* in $\mathbf{Set}(\mathbf{2})$ of the syntactic theory $\mathsf{SynLP}_{\mathbb{L}}$.

3.3 A Gallery of diagrammatic representations of semantic constructs

A pleasant outcome of the functorial semantics is the possibility of capturing some well-known semantic approaches to logic programs using the diagrammatic language. More precisely, we will study string diagrams in the syntax category whose images under the model are the semantics we are interested in. We shall discuss three of them: the immediate consequence operator, the least Herbrand semantics, and the stratified semantics.

Immediate consequence operator

We begin with the immediate consequence operator $\mathbf{T}$ (*cf. Section 2*), a basic yet fundamental concept beneath many denotational semantics of logic programs [17, 3, 18, 38] . Let us fix a logic program $\mathbb{P}$ with atoms $At = \{A_1, \dots, A_n\}$ and $\mathbb{L} := [\mathbb{P}]$. Intuitively, given an interpretation $\mathcal{I}$, $\mathbf{T}_{\mathbb{P}}(\mathcal{I})$ is the set of atoms derivable from the set $\mathcal{I}$ of 'assumptions' using each $\mathbb{P}$-clause exactly once, in parallel. Note that $\mathbf{T}_{\mathbb{P}}$ is not extensive, in the sense that $A \in \mathcal{I}$ does not imply $A \in \mathbf{T}_{\mathbb{P}}(\mathcal{I})$. Also, note there is a canonical isomorphism between $\mathcal{P}(At)$ and $\mathbf{2}_{A_1} \times \cdots \times \mathbf{2}_{A_n}$, which we will exploit to formulate the operator as a morphism in $\mathbf{Set}(\mathbf{2})$.

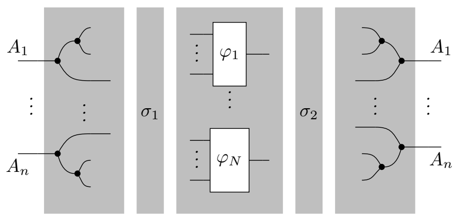
▶ **Example 6.** We consider an example from [18]. The atom set At_{pq} consists of $p(1,1)$, $p(1,2)$, $p(2,1)$, $p(2,2)$, $q(1)$, $q(2)$, and the program $\mathbb{P}_{pq}$ contains the following six clauses:

$$\begin{array}{lll} \psi_1 \equiv p(1,2) \leftarrow . & \psi_3 \equiv q(1) \leftarrow p(1,1), \neg q(1). & \psi_5 \equiv q(2) \leftarrow p(2,1), \neg q(1). \\ \psi_2 \equiv p(2,1) \leftarrow . & \psi_4 \equiv q(1) \leftarrow p(1,2), \neg q(2). & \psi_6 \equiv q(2) \leftarrow p(2,2), \neg q(2). \end{array}$$

Then $\mathbf{T}_{\mathbb{P}_{pq}}$ is a function $\mathcal{P}(At_{pq}) \to \mathcal{P}(At_{pq})$ which, for instance, maps both $\varnothing$ and $\{p(1,2), q(2)\}$ to $\{p(1,2), p(2,1)\}$, and maps $\{p(1,2)\}$ to $\{p(1,2), p(2,1), q(1)\}$.

We can express $\mathbf{T}_{\mathbb{P}}$ via our diagrammatic language by putting side-by-side all the generating morphisms in $\Sigma_{\mathbb{L}}$, one for each clause.

▶ **Proposition 7.** *The following string diagram $t_{[\mathbb{P}]}$ (in $\mathsf{SynLP}_{[\mathbb{P}]}$) satisfies $[\![t_{[\mathbb{P}]}]\!]_{\mathbb{P}} = \mathbf{T}_{\mathbb{P}}$.*



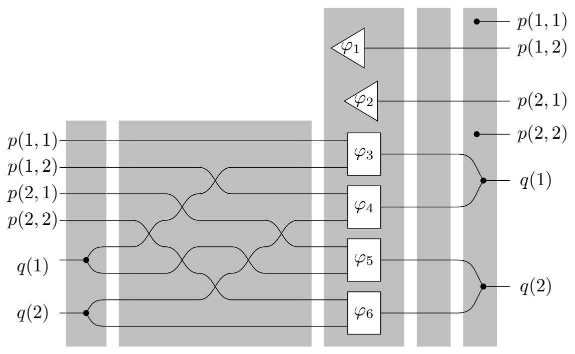
(4)

The left and the right ports both consist of all the atoms $A_1, \dots, A_n$ in At. For each A_i, suppose there are k_i-many clauses whose bodies include A_i and ℓ_i-many clauses whose heads are A_i. We make k_i copies (via -•< in the left-most box) and ℓ_i cocopies (via >•- in the right-most box) of A_i. The middle box contains the parallel composition of all $\mathbb{L}$-clauses $\varphi_1, \dots, \varphi_N$. In the two σ_i-boxes, we have suitably many swapping morphisms $\times$ to match each copy/cocopy of A_i with an input/output wire A_i in the middle box.

▶ **Example 8.** The program $\mathbb{P}_{pq}$ in Example 6 consists of six definite clauses:

$$\begin{array}{lll} \varphi_1 \equiv p(1,2) \leftarrow . & \varphi_3 \equiv q(1) \leftarrow p(1,1), q(1). & \varphi_5 \equiv q(2) \leftarrow p(2,1), q(1). \\ \varphi_2 \equiv p(2,1) \leftarrow . & \varphi_4 \equiv q(1) \leftarrow p(1,2), q(2). & \varphi_6 \equiv q(2) \leftarrow p(2,2), q(2). \end{array}$$

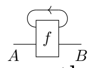
These clauses also constitute the set of generating morphisms $\Sigma_{\mathbb{P}_{pq}}$ of $\mathsf{SynLP}_{\mathbb{P}_{pq}}$. Then the corresponding string diagram $t_{[\mathbb{P}_{pq}]}$ for $\mathbf{T}_{\mathbb{P}_{pq}}$ is:



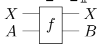
Interlude: traced extension

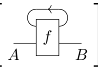
The last two semantic constructs we will consider are *fixed point-style* semantics: to provide the same kind of analysis as for the immediate consequence operator, we need to mildly extend our string diagrammatic language to include 'feedback wires' (called traces), and also extend the semantic category accordingly. The full picture of this extension is in (5) below.

▶ Remark 9. We purposefully choose not to include traces in the 'basic' syntax category (Sec. 3.1), as we consider fixed points a feature specific to certain semantic constructs, rather than a primitive construction of logic program syntax. Also, note that traces are not required for our next developments in case one restricts attention to *acyclic* programs. This is mostly evident in the probabilistic case, where the distribution semantics is only defined for acyclic programs, and indeed the modelisation only uses the basic syntax, without traces.

We briefly recall traced categories, referring to [2] for full details. Recall that a symmetric monoidal category $\langle \mathbf{C}, \otimes, I\rangle$ is *traced* if it is equipped with a natural family of functions $Tr^X_{A,B}(f) \colon \mathbf{C}(X \otimes A, X \otimes B) \to \mathbf{C}(A, B)$ satisfying certain compatibility conditions [2]. In string diagrams, $Tr^X_{A,B}(f)$ is depicted as 

. We will use the free construction freeTrCDMU(X, Σ) of a traced CDMU category: the objects are finite lists over X, the morphisms are obtained as string diagrams in freeCDMU(X, Σ) plus the possibility of adding feedback loops, modulo the axioms for traced categories (*cf.* [1] for a detailed definition). The syntax category for $\mathbb{P}$ is defined as freeTrCDMU$(At, \Sigma_{\mathbb{L}})$, where $\mathbb{L} = [\mathbb{P}]$. As for the semantics categories, we move to the category $\mathbf{Rel}(\mathbf{2})$ whose objects are the same as $\mathbf{Set}(\mathbf{2})$ and morphisms $A \to B$ are *relations* $R \subseteq A \times B$. In fact, $\mathbf{Rel}(\mathbf{2})$ is a compact closed category — where we let the cartesian product act as the monoidal product — and thus equipped with a canonical traced structure (inherited from the category of relations).

Each logic program $\mathbb{P}$ uniquely determines a traced CDMU functor $[\![-]\!]^{Tr}_{\mathbb{P}}$, inductively defined on the generating morphisms as follows: for every trace-free string diagram f, $[\![f]\!]^{Tr}_{\mathbb{P}}$ is the relation describing the graph of the function $[\![f]\!]_{\mathbb{P}}$; for traced diagrams, given 

, $\left[\!\!\left[\right.\right.$ 

$\left.\left.\right]\!\!\right]^{Tr}_{\mathbb{P}}$ is $\{(a,b) \in [\![A]\!]^{Tr}_{\mathbb{P}} \times [\![B]\!]^{Tr}_{\mathbb{P}} \mid \exists x \in [\![X]\!]^{Tr}_{\mathbb{P}} : ((x,a),(x,b)) \in [\![f]\!]^{Tr}_{\mathbb{P}}\}$.

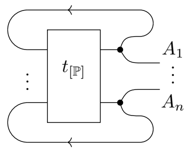
The relationship between $[\![-]\!]_{\mathbb{P}}$ and $[\![-]\!]^{Tr}_{\mathbb{P}}$ can be summarised as the commutative square (5), where i identifies $\mathsf{SynLP}_{\mathbb{L}}$ as a subcategory of $\mathsf{SynLP}^{Tr}_{\mathbb{L}}$, and ι maps a function to its graph.

$$\begin{tikzcd}
 & \mathsf{SynLP}^{Tr}_{\mathbb{L}} \arrow[dr, "{[\![-]\!]^{Tr}_{\mathbb{P}}}"] & \\
\mathsf{SynLP}_{\mathbb{L}} \arrow[ur, "i"] \arrow[dr, "{[\![-]\!]_{\mathbb{P}}}"'] & & \mathbf{Rel}(\mathbf{2}) \\
 & \mathbf{Set}(\mathbf{2}) \arrow[ur, "\iota"'] &
\end{tikzcd} \quad (5)$$

Herbrand semantics

The importance of the immediate consequence operator $\mathbf{T}_{\mathbb{P}}$ lies in that it performs the single 'iteration step' in various denotational semantics of logic programs. For definite logic programs, the least models are precisely the least fixed points of the immediate consequence operator, which exist because the operators are monotonic on the complete lattice $\mathcal{P}(At)$. This suggests that one can express the *least Herbrand model* $\mathcal{H}$ (*cf.* Section 2) simply as immediate consequence operator plus 'feedback wires'.

▶ **Proposition 10.** *Suppose $\mathbb{P}$ is definite. The string diagram $h_{[\mathbb{P}]}$ in $\mathsf{Syn}^{cb}_{[\mathbb{P}]}$ defined in (6) expresses the least Herbrand model of $\mathbb{P}$, in the sense that $\mathcal{H}(\mathbb{P}) = \min \llbracket h_{[\mathbb{P}]} \rrbracket^{Tr}_{\mathbb{P}}$, where the box is (4).*



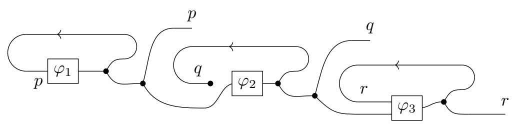
(6)

Observe that, even though the least Herbrand model is just one element of $\llbracket h_{[\mathbb{P}]} \rrbracket^{Tr}_{\mathbb{P}}$ in (6), as a whole $\llbracket h_{[\mathbb{P}]} \rrbracket^{Tr}_{\mathbb{P}}$ still expresses a well-known semantic concept, namely the set of *supported models* of the logic program $\mathbb{P}$. An interpretation $\mathcal{I}$ is a *model* of $\mathbb{P}$ if for each $\mathbb{P}$-clause ψ, $\mathcal{I} \vDash \mathsf{body}(\psi)$ implies $\mathsf{head}(\psi) \in \mathcal{I}$; it is *supported* if for each $\mathbb{P}$-clause ψ, $\mathsf{head}(A) \in \mathcal{I}$ implies $\mathcal{I} \vDash \mathsf{body}(\psi)$. Note that supported models are exactly the fixed points of $\mathbf{T}_{\mathbb{P}}$ [33]. It then follows immediately that the string diagram $h_{[\mathbb{P}]}$ (6) expresses the supported models of $\mathbb{P}$: $\llbracket h_{[\mathbb{P}]} \rrbracket^{Tr}_{\mathbb{P}} = \{\mathcal{I} \mid \mathcal{I} \text{ is a supported model of } \mathbb{P}\}$.

Stratified semantics

We move on to the case of stratified semantics of stratified logic programs (*cf.* Section 2). Suppose program $\mathbb{P}$ is stratified, where $At_1, \cdots, At_k$ is a stratification of At, and $\mathbb{P}_1, \ldots, \mathbb{P}_k$ is the associated partition of $\mathbb{P}$. The idea of stratification is to turn $\mathbb{P}_1, \ldots, \mathbb{P}_k$ into definite logic programs whose least models together form a model of the original program $\mathbb{P}$. As recalled in Section 2, this is achieved by observing that $\mathbb{P}_1$ is always definite by definition, and $\mathbb{P}_i$ for $i > 1$ can be turned into a definite program using the least models for the definite programs of $\{\mathbb{P}_j\}_{j<i}$. This idea suggest the construction of a string diagram representing stratified semantics in a layer-by-layer style: the output wires of all $\{h_{[\mathbb{P}_j]}\}_{j<i}$ (defined in Proposition 10) will serve as inputs of $h_{[\mathbb{P}_i]}$. For the sake of clarity, rather than detailing the fully general case of this construction, we believe it is best illustrated via an example.

▶ **Example 11.** Consider the program $\mathbb{O}$ consisting of clauses $\psi_1 \equiv p \leftarrow p$, $\psi_2 \equiv q \leftarrow \neg p$ and $\psi_3 \equiv r \leftarrow r, \neg q$. One stratification is $P_1 = \{p\}$, $P_2 = \{q\}$, $P_3 = \{r\}$, whose corresponding partition of $\mathbb{O}$ is $\mathbb{O}_1 = \{p \leftarrow p.\}$, $\mathbb{O}_2 = \{q \leftarrow \neg p.\}$, $\mathbb{O}_3 = \{r \leftarrow r, \neg q.\}$. Let φ_i be $\tau^{gen}(\psi_i)$, for $i = 1, 2, 3$.
The string diagram $s_{[\mathbb{O}]}$ on the right expresses the stratified model $\{q\}$ of $\mathbb{O}$, in the sense that the stratified model is the least element under the lexicographical order in $\llbracket s_{[\mathbb{O}]} \rrbracket_{\mathbb{O}}$: for any



$(i_p, i_q, i_r) \in \llbracket s_{[\mathbb{O}]} \rrbracket_{\mathbb{O}}$, there are three possibilities, either $\neg p \leq i_p$, or $\neg p = i_p$ and $q \leq i_q$, or $\neg p = i_p$, $q = i_q$ and $\neg r \leq i_z$.

4 Probabilistic Logic Programming

In this section we turn to probabilistic logic programming (PLP). We first give the functorial semantics of PLP in Section 4.1. Next we discuss a pictorial representation of distribution semantics, in Section 4.2. A benefit of the diagrammatic representation of PLP syntax as string diagrams is that the correspondence with a closely related formalism, namely *Bayesian networks* (BNs), become more apparent: we explore this in Section 4.3, where we show the equivalence between boolean-valued BNs and acyclic PLP and illustrate their relationship with functors between the corresponding categories.

For PLP, the separation of syntax and semantics into different categories provides two main insights: first, we are able to define PLP programs as models of the same syntax category as LP programs, thus formalising the intuition that the difference between the two formalisms is only at the semantics level. Second, in order to formalise the correspondence between PLP programs and boolean-valued BNs, we only need to act on the syntactic categories for the two formalisms, thus showing that the two rely on the same semantic layer, and differ in how this is described by syntactic/combinatorial structures.

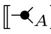
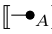
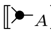
4.1 Functorial Semantics of PLP

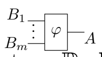
Throughout this section we fix an acyclic definite logic program $\mathbb{L}$. To be precise, by '$\mathbb{L}$ being acyclic' we mean that there is no finite sequence of $\mathbb{L}$-clauses $\varphi_0, \ldots, \varphi_m$ such that $\mathsf{head}(\varphi_{i+1}) \in \mathsf{body}(\varphi_i)$ for all $i = 0, \ldots, m-1$, and $\mathsf{head}(\varphi_0) \in \mathsf{body}(\varphi_m)$. Our goal is to provide for PLP a functorial semantics characterisation analogous to the one of Proposition 5. As mentioned above, we will use the same syntax category as for LP, introduced in Section 3.1. What differs is the semantics category, which we now introduce. Intuitively, it amounts to switching from boolean-valued functions to their probabilistic counterpart.

▶ **Definition 12.** *The category* $\mathbf{Stoch}(\mathbf{2})$ *is defined as having the same objects as* $\mathbf{Set}(\mathbf{2})$*, and morphisms the functions of the form* $f\colon \mathbf{2}_{A_1} \times \cdots \times \mathbf{2}_{A_k} \to \mathcal{D}(\mathbf{2}_{B_1} \times \cdots \times \mathbf{2}_{B_m})$*, where* $\mathcal{D}(\mathbf{2}_{B_1} \times \cdots \times \mathbf{2}_{B_m})$ *is the set of probability distributions on* $\mathbf{2}_{B_1} \times \cdots \times \mathbf{2}_{B_m}$*. Given morphisms* $f\colon X \to \mathcal{D}(Y)$ *and* $g\colon Y \to \mathcal{D}(Z)$*, their composition* $g \circ f$ *assigns to each* $x \in X$ *a distribution* $\sum_{z \in Z} \left(\sum_{y \in Y} f(x)(y) \cdot g(y)(z) \right) |z\rangle$*.*

Equivalently, $\mathbf{Stoch}(\mathbf{2})$ is the full subcategory of $\mathbf{Stoch}$ (the category of stochastic matrices) whose objects are those of $\mathbf{Set}(\mathbf{2})$. Also, $\mathbf{Stoch}$ is the same as the Kleisli category for the distribution monad, so we may regard composition in $\mathbf{Stoch}(\mathbf{2})$ simply as Kleisli composition.

Given a PLP $\mathbb{P}$ with $[\mathbb{P}] = \mathbb{L}$, we define a functor $[\![-]\!]_{\mathbb{P}} : \mathsf{SynLP}_{\mathbb{L}} \to \mathbf{Stoch}(\mathbf{2})$ as follows. On objects, $[\![-]\!]_{\mathbb{P}}$ maps $A \in At$ to $\mathbf{2}_A$. On morphisms, for the CDMU structure we define

$[\![$$_A]\!]_{\mathbb{P}} : \mathbf{2}_A \to \mathbf{2}_A \times \mathbf{2}_A \quad x \mapsto 1|(x,x)\rangle$ $\qquad [\![$$_A]\!]_{\mathbb{P}} : \mathbf{2}_A \to \mathbf{1} \quad x \mapsto 1|*\rangle$ $\qquad [\![$$_A]\!]_{\mathbb{P}} : \mathbf{2}_A \times \mathbf{2}_A \to \mathbf{2}_A \quad (x,y) \mapsto 1|x \vee y\rangle$ $\qquad [\![$ $_A]\!]_{\mathbb{P}} : \mathbf{1} \to \mathbf{2}_A \quad * \mapsto 1|\neg A\rangle$

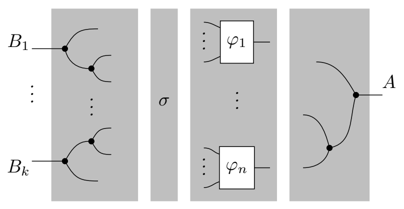
For each generating morphism  in $\mathsf{SynLP}_{\mathbb{L}}$, $[\![\varphi]\!]_{\mathbb{P}}$ maps a state $u \in \mathbf{2}_{B_1} \times \cdots \times \mathbf{2}_{B_m}$ to $p|A\rangle + (1-p)|\neg A\rangle$ if there exists a $\mathbb{P}$-clause ψ such that $\tau^{pr}(\psi) = \varphi$ and $lab(\psi) = p$, and to $1|\neg A\rangle$ otherwise. As in the classical case, for $[\![-]\!]_{\mathbb{P}}$ to be well-defined, we need the semantics category $\mathbf{Stoch}(\mathbf{2})$ to satisfy the CDMU equations as shown in Appendix B, Lemma 28. We also refer to Appendix B for the proof of our characterisation result, which essentially follows the same steps as the proof of Proposition 5.

▶ **Proposition 13.** *There is a 1-1 correspondence between PLP programs based on* $\mathbb{L}$ *and generator-preserving CDMU functors* $\mathsf{SynLP}_{\mathbb{L}} \to \mathbf{Stoch}(\mathbf{2})$*.*

4.2 Distribution Semantics in String Diagrams

As for LP, we now study diagrammatic representations of PLP semantics. As a preliminary, we record the notion of 'A-component'. Intuitively the A-component collects all the clauses with heads A, and 'bundle' them into a string diagram with a single output A.

▶ **Definition 14.** *The A-component of $\mathbb{P}$ (based on $\mathbb{L}$) is a string diagram $comp_A : [B_1, \dots, B_k] \to [A]$ in $\mathsf{SynLP}_{\mathbb{L}}$ constructed as on the left below, where: (I) in the first block, for each atom B_i appearing in the body of some φ with $\mathsf{head}(\varphi) = A$, there are k_i-many copies of B_i (via —<), where k_i is the number of $\mathbb{L}$-clauses φ satisfying $\mathsf{head}(\varphi) = A$ and $B \in \mathsf{body}(\varphi)$.*

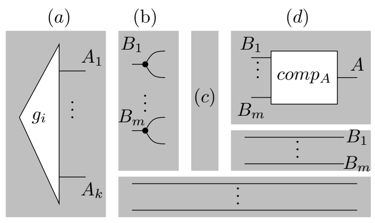


(II) In the third block, we have parallel string diagrams for each $\mathbb{L}$-clause $\varphi_1, \dots, \varphi_n$ with head A.
(III) The fourth block hosts n-many cocopies of A (via >—), where n is the number of $\mathbb{L}$-clauses with head A.
(IV) the σ-block contains suitably many swapping morphisms χ to match each copy of B_i in the first block to an input wire B_i in the third block.
Note that, if $n = 0$ (namely there is no $\mathbb{L}$-clause with head A), then $comp_A$ simplifies to •—.

Now the goal is to diagrammatically express the probability distribution $\delta(\bar{A})$ of a set of atoms $\bar{A} := \{A_1, \cdots, A_k\}$. The idea is to construct a string diagram $f_{\bar{A}} : [\,] \to [A_1, \dots, A_k]$ which exhausts all possible derivations of $\bar{A}$ in $\mathbb{P}$. To do so, we will first define f_{At} and then obtain $f_{\bar{A}}$ by discarding all the atoms not appearing in $\bar{A}$. Note that the termination of the following construction relies on $\mathbb{P}$ being acyclic, and $comp_A$ is defined in Definition 14.

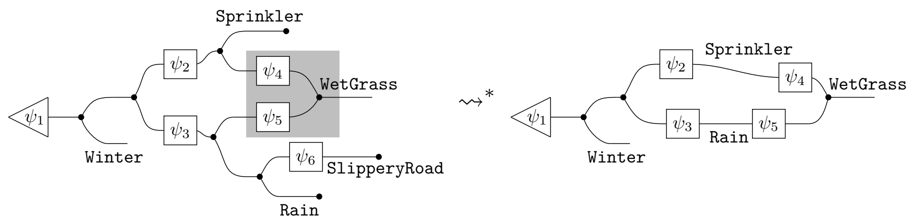
▶ **Construction 15.** Suppose $At = \{A_1, \dots, A_n\}$. The string diagram $f_{At} : [\,] \to [A_1, \dots, A_n]$ is defined via a step-by-step construction of morphisms g_i for $i \in \mathbb{N}$:

1. $g_0 = id_{[\,]} : [\,] \to [\,]$.
2. If g_i is of type $[\,] \to [A_1, \dots, A_n]$, then let $f_{At} = g_i$.
3. Otherwise, g_i is of type $[\,] \to [A_1, \dots, A_k]$, and we define g_{i+1} as the string diagram on the right. We pick any $A \in At \setminus \{A_1, \dots, A_k\}$ such that all atoms in the domain of $comp_A$ already appear in $A_1, \dots, A_k$, say $comp_A : [B_1, \dots, B_m] \to [A]$ with $\{B_1, \dots, B_m\} \subseteq \{A_1, \dots, A_k\}$. Then in block (b) we make 2 copies for each $B_j \in \{B_1, \dots, B_m\}$ from the codomain of g_i in block (a), compose them with suitably many swapping morphisms χ in block (c), and match one copy of each B_j to exactly one B_j from the domain of $comp_A$ in block (d).



We are now ready to construct the diagram $f_{\bar{A}}$. First, for each atom $B \in At \setminus \{A_1, \dots, A_k\}$, discard B in f_{At} by post-composing —•$_B$. Next, simplify the diagram by the rewriting rules —< ⇝ —, —< ⇝ —, and —$\boxed{f}$• ⇝ —•. Note that while the first two rewriting rules are CDMU axioms, the third rewriting rule is not valid in $\mathsf{SynBN}_{\mathbb{L}}$. However, all of them are valid in the semantics category $\mathbf{Stoch(2)}$ [22], which justifies our procedure.

▶ **Example 16.** Recall the program $\mathbb{P}_{\text{wet}}$ from Example 1. The string diagram below left is $f_{At_{\text{wet}}}$ with atoms `Sprinkler`, `Rain` and `SlipperyRoad` discarded.



In particular, the subdiagram in the grey block is the `WetGrass`-component of type $[\texttt{Sprinkler}, \texttt{Rain}] \to [\texttt{WetGrass}]$. The string diagram $f_{\bar{A}}$ below right is the result of applying the three rewriting rules. One can verify that $[\![f_{\bar{A}}]\!]_{\mathbb{P}_{\text{wet}}}$ is the probability distribution $\delta(\texttt{Winter}, \texttt{WetGrass})$.

▶ **Proposition 17.** *$f_{\bar{A}}$ calculates the success probability: $[\![f_{\bar{A}}]\!]_{\mathbb{P}} = \delta(\bar{A})$.*

4.3 Correspondence of PLP and BNs via Functorial Semantics

We now turn attention to the relationship between PLP and Bayesian networks. Our goal is formulating the translation between the two formalisms at the functorial level, taking advantage on the one side of the functorial semantics of PLP just established, and on the other side of the functorial semantics of Bayesian networks, as described in [22].

It is known in the LP literature that every acyclic PLP can be transformed into an equivalent boolean-valued BN, and vice versa [31]. Intuitively, starting from a BN $\mathbb{B} = (G, \mathrm{Pr})$ (*cf.* Section 2), one constructs a PLP program with a clause for each conditional probability in Pr. Conversely, given a PLP program $\mathbb{P}$, one constructs a DAG in which B is a parent of A precisely when there *exists* $\psi \in \mathbb{P}$ such that $A = \mathsf{head}(\psi)$ and B appears in $\mathsf{body}(\psi)$. Each conditional probability $\mathrm{Pr}(A = 1 \mid pa(A) = u)$ is calculated by 'summing up' the probabilities of all $\mathbb{B}$ clauses ψ such that $\mathsf{head}(\psi) = A$ and $u \vDash \mathsf{body}(\psi)$ as independent random events.

▶ **Example 18.** Consider the PLP $\mathbb{P}_{\text{wet}}$ from Example 1. On the left below we show the DAG of the corresponding Bayesian network, and on the right the conditional probability associated with variable `WetGrass`.

$$\begin{tikzcd}
& \boxed{\texttt{Winter}} \arrow[dl] \arrow[dr] & \\
\boxed{\texttt{Rain}} \arrow[d] \arrow[dr] & & \boxed{\texttt{Sprinkler}} \arrow[dl] \\
\boxed{\texttt{SlipperyRoad}} & \boxed{\texttt{WetGrass}} &
\end{tikzcd}$$

		Pr(`WetGrass`)
¬`Sprinkler`	¬`Rain`	0
¬`Sprinkler`	`Rain`	0.8
`Sprinkler`	¬`Rain`	0.9
`Sprinkler`	`Rain`	0.98

(7)

Conversely, we may construct a PLP $\mathbb{P}'_{\text{wet}}$ corresponding to $\mathbb{B}_{\text{wet}}$, following the recipe given above. It differs with $\mathbb{P}_{\text{wet}}$ only on the clauses with heads `WetGrass`. Instead of ψ_5 and ψ_6 in Example 1, $\mathbb{P}'_{\text{wet}}$ has the following clauses which together specify the conditional probability in Figure (7):

$$\begin{array}{rlll} 0 :: & \texttt{WetGrass} & \leftarrow \neg\texttt{Sprinkler}, \neg\texttt{Rain}. & \quad 0.8 :: \ \texttt{WetGrass} \leftarrow \neg\texttt{Sprinkler}, \texttt{Rain}. \\ 0.9 :: & \texttt{WetGrass} & \leftarrow \texttt{Sprinkler}, \neg\texttt{Rain}. & \quad 0.98 :: \ \texttt{WetGrass} \leftarrow \texttt{Sprinkler}, \texttt{Rain}. \end{array}$$

We now wish to study this two-way translation at the functorial level. To do so, we exploit the functorial semantics for BNs, as established in [22] (see also [16]), and reported in Appendix A. In a nutshell, this characterisation associates Bayesian networks based on a DAG $G = (V_G, \Sigma_G)$ with models of a freely generated syntax category $\mathsf{SynBN}_G \coloneqq \mathsf{freeCD}(V_G, \Sigma_G)$, where intuitively the edges of G act as the generators of the diagrammatic syntax. For comparing it to PLP, we need a restriction of this characterisation result to *boolean-valued* Bayesian networks (*cf.* Section 2), which we report below.

▶ **Proposition 19** ([22])**.** *There is 1-1 correspondence between boolean-valued Bayesian networks based on a DAG G and generator-preserving CD functors $\mathsf{SynBN}_G \to \mathbf{Stoch}(\mathbf{2})$.*

We now describe the two-way translation between PLP and BNs, exploiting their view as models as provided by Propositions 13 and 19.

BN to PLP. Given a boolean-valued BN $\mathbb{B} = (G, \mathrm{Pr})$ and its corresponding BN model $[\![-]\!]_{\mathbb{B}} : \mathsf{SynBN}_G \to \mathbf{Stoch}(\mathbf{2})$, we let the syntax category $\mathsf{SynLP}_{\mathbb{L}'}$ of the corresponding logic program be $\mathsf{freeCDMU}(V_G, \Sigma_G)$. Intuitively, this means that every node A in G yields exactly one clause $A \leftarrow pa(A)$ in $\mathbb{L}'$. Then we can define a PLP program $\mathbb{P}'$ via Proposition 13 as the CDMU functor $[\![-]\!]^b_{\mathbb{B}} : \mathsf{SynLP}_{\mathbb{L}'} \to \mathbf{Stoch}(\mathbf{2})$ obtained by canonically lifting the CD functor $[\![-]\!]_{\mathbb{B}} : \mathsf{SynBN}_G \to \mathbf{Stoch}(\mathbf{2})$ along the inclusion functor $\iota \colon \mathsf{SynBN}_G \to \mathsf{SynLP}_{\mathbb{L}'}$ which embeds the CD structure of $\mathsf{SynBN}_G = \mathsf{freeCD}(V_G, \Sigma_G)$ into the CDMU structure of $\mathsf{SynLP}_{\mathbb{L}'} = \mathsf{freeCDMU}(V_G, \Sigma_G)$. In concrete, $[\![-]\!]^b_{\mathbb{B}}$ maps $\succ\!\!\bullet\!-$, $\bullet\!-$ to the comonoid structure in $\mathbf{Stoch}(\mathbf{2})$ and just behaves the same as $[\![-]\!]_{\mathbb{B}}$ on the rest. The construction of $[\![-]\!]^b_{\mathbb{B}}$ is summarised by the commutative square ② in (9) below. We can verify that this lifting $[\![-]\!]^b_{\mathbb{B}}$ indeed coincides with the model $[\![-]\!]_{\mathbb{P}'} : \mathsf{SynLP}_{\mathbb{L}'} \to \mathbf{Stoch}(\mathbf{2})$ associated with $\mathbb{P}'$, the PLP program that one could obtain from $\mathbb{B}$ in the traditional way (e.g. in [31]).

▶ **Proposition 20.** *Let $\mathbb{P}'$ be the PLP program encoding $\mathbb{B}$. Then $\Sigma_{[\mathbb{P}']} = \Sigma_{\mathbb{L}'}$ and $[\![-]\!]_{\mathbb{P}'} = [\![-]\!]^b_{\mathbb{B}}$.*

PLP to BN. We turn to the converse direction: starting from a PLP model $[\![-]\!]_{\mathbb{P}} : \mathsf{SynLP}_{\mathbb{L}} \to \mathbf{Stoch}(\mathbf{2})$, we construct a BN model $[\![-]\!]_{\mathbb{B}}$. The main insight is that this construction is purely syntactical: the hurdle to take is figuring out the correct syntax category Σ_H, and a 'syntax translation' functor $\mathcal{F} \colon \mathsf{SynBN}_H \to \mathsf{SynLP}_{\mathbb{L}}$ (note the direction!). Then the BN will be the model defined by the composite functor $[\![-]\!]_{\mathbb{P}} \circ \mathcal{F} \colon \mathsf{SynBN}_H \to \mathbf{Stoch}(\mathbf{2})$.

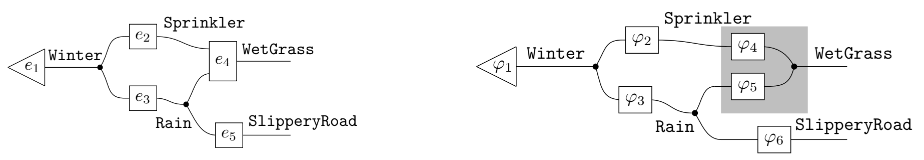
First, we define generating objects V_H and generating morphisms Σ_H yielding our syntax category $\mathsf{SynBN}_G := \mathsf{freeCD}(V_H, \Sigma_H)$. We let $V_H = At$, and Σ_H be

$$\left\{ \begin{smallmatrix} B_1 \\ \vdots \\ B_m \end{smallmatrix} \boxed{e} \, A \;\middle|\; \{B_1, \ldots, B_m\} = \{B \mid \exists \varphi \in \mathbb{L} \text{ such that } A = \mathsf{head}(\varphi), B \in \mathsf{body}(\varphi)\} \right\}$$

Intuitively, inputs of e are obtained by combining all $\mathbb{L}$-clauses with the same head A. We now define the 'translation' $\mathcal{F} \colon \mathsf{SynBN}_H \to \mathsf{SynLP}_{\mathbb{L}}$. The idea is that $\mathcal{F}$ 'decomposes' children and their parents into $\mathbb{L}$-clauses. Formally, it is the identity on objects and on morphisms it is freely defined by the following mapping on the generating morphisms of SynBN_H:

- For $d \in \{-\!\!\prec, -\!\bullet\}$, $\mathcal{F}(d) := d$.
- For each generating morphism $\begin{smallmatrix} B_1 \\ \vdots \\ B_m \end{smallmatrix} \boxed{e} \, A$ in Σ_H, $\mathcal{F}(e) := comp_A$ (see Definition 14).

▶ **Example 21.** Recall $\mathbb{P}_{\text{wet}}$ from Example 1. Applying $\mathcal{F}_{\text{wet}} \colon \mathsf{SynBN}_{G_{\text{wet}}} \to \mathsf{SynLP}_{\mathbb{P}_{\text{wet}}}$ to the string diagram below left (*cf.* the DAG of Example 18) yields the string diagram below right, where the φ_is are as in Example 3.



(8)

Moreover, the $[\![-]\!]_{\mathbb{P}} \circ \mathcal{F}_{\text{wet}}$-image of the string diagram in the grey block yields precisely the conditional probability distribution $\mathbf{2}_{\texttt{Sprinkler}} \times \mathbf{2}_{\texttt{Rain}} \to \mathcal{D}(\mathbf{2}_{\texttt{WetGrass}})$ in $\mathbb{B}_{\text{wet}}$ represented by the conditional probability distribution in (7).

▶ Remark 22. Thanks to the separation of syntax and semantics, our construction is able to simplify and divide in two steps what in the literature (*cf.* [31]) is performed in a single step. In the traditional approach, from $\mathbb{P}$ a DAG is constructed with 'AND' nodes and 'noisy-OR' nodes [31], from which one derives the conditional probability distributions. Instead, we construct a simpler DAG $H = (V_H, \Sigma_H)$ — in fact, a syntax category SynBN_H encoding H — and only as a next step we introduce a richer structure (modelled with $\succ\!\!\bullet\!-$, see (8)) via $\mathcal{F}_{\text{wet}}$. Moreover, all these steps are performed at a purely syntactic level: obtaining the conditional probabilities is 'delegated' to composition with the given functor $[\![-]\!]_{\mathbb{P}} : \mathsf{SynLP}_{\mathbb{L}} \to \mathbf{Stoch}(\mathbf{2})$.

As with the converse direction, we may verify (see Appendix B) that the BN derived from $\mathbb{P}$ in the traditional way (*cf.* [31]) coincides with the one obtained via our construction.

▶ **Proposition 23.** *Let $\mathbb{B} = (G, \Pr)$ be the Bayesian network constructed from $\mathbb{P}$, then $\Sigma_G = \Sigma_H$, and $[\![-]\!]_{\mathbb{B}} = [\![-]\!]_{\mathbb{P}} \circ \mathcal{F}$.*

▶ Remark 24. Note that, unlike Proposition 23, in Proposition 20 we cannot obtain the PLP model $[\![-]\!]_{\mathbb{P}}$ as the composition of the BN model $[\![-]\!]_{\mathbb{B}}$ and a functor between syntax categories. This is because we lack a functor $\mathsf{SynLP}_{\mathbb{L}} \to \mathsf{SynBN}_{\mathbb{B}}$: $\mathsf{SynLP}_{\mathbb{L}}$ has a richer structure than $\mathsf{SynBN}_{\mathbb{B}}$, and there is no canonical way to map the monoid structure ${>}\!\!\bullet\!\!{-}, \bullet\!\!{-}$.

The two constructions of this section are summarised by the following commutative diagram:

$$\begin{tikzcd}
\mathsf{SynLP}_{\mathbb{L}} \arrow[drr, "{[\![-]\!]_{\mathbb{P}}}"'] \arrow[dr, phantom, "①"] & & \mathsf{SynBN}_G \arrow[ll, "\mathcal{F}"'] \arrow[rr, hook, "\iota"] \arrow[d, "{[\![-]\!]_{\mathbb{B}}}"'] \arrow[r, phantom, "③"', yshift=15pt] & & \mathsf{SynLP}_{\mathbb{L}'} \arrow[llll, bend right=30, "\mathcal{G}"'] \arrow[d, "{[\![-]\!]_{\mathbb{P}'}}"] \\
& & \mathbf{Stoch}(\mathbf{2}) \arrow[rr, equal] \arrow[u, phantom, ""] & \arrow[u, phantom, "②"] & \mathbf{Stoch}(\mathbf{2})
\end{tikzcd} \tag{9}$$

Given the definition of SynBN_G and $\mathsf{SynLP}_{\mathbb{L}'}$, we may let $\mathcal{G}$ be $\mathcal{F}$ plus the clause that $\mathcal{G}(d) = d$ for $d \in \{{>}\!\!\bullet\!\!{-}, \bullet\!\!{-}\}$. Proposition 23 amounts to commutativity of ①, and ② states that $[\![-]\!]_{\mathbb{B}}$ factors through its CDMU-lifting $[\![-]\!]_{\mathbb{P}'}$ via the inclusion functor ι. ③ connects the program $\mathbb{P}'$ obtained from $\mathbb{B}$ with the original program $\mathbb{P}$ (where $\mathbb{L} = [\mathbb{P}]$ and $\mathbb{L}' = [\mathbb{P}']$).

5 Weighted Logic Programming

We conclude by extending our approach to encompass weighted logic programming (WLP), a generalisation of logic programming for specifying dynamic programming algorithms [14, 10, 15]. We will provide a functorial semantics for WLP, based on which one can express the standard semantics of WLP diagrammatically.

The syntax category is the same as for LP and PLP, reflecting the intuition that the extension provided by WLP only affects the semantics layer. The semantics category for WLP should reflect the fact that atoms are now interpreted no longer as boolean values but as values in a given semiring $\mathcal{K}$. Interestingly, the variation required does not change the morphisms of the 'basic' semantics category $\mathbf{Set}(\mathbf{2})$, as in the PLP case, but the objects. Whereas the semantics category $\mathbf{Stoch}(\mathbf{2})$ for PLP can be thought as a 'Kleisli' variation of $\mathbf{Set}(\mathbf{2})$, the semantics category $\mathbf{Set}(\mathcal{K})$ for WLP simply changes the generating objects of $\mathbf{Set}(\mathbf{2})$ from copies of the Boolean algebra $\mathbf{2}$ to copies of the semiring $\mathcal{K}$.

More precisely, we define $\mathbf{Set}(\mathcal{K})$ as the category whose objects are finite products of the form $\mathcal{K}_{A_1} \times \cdots \times \mathcal{K}_{A_k}$ (*cf. Section 2*). In particular, the singleton set $\mathbf{1}$ is the empty product. The $\mathbf{Set}(\mathcal{K})$-morphisms are functions between the underlying sets. Now, every WLP program $\mathbb{P}$ with atom set At and $\mathbb{L} := [\mathbb{P}]$ uniquely determines a functor $[\![-]\!]_{\mathbb{P}} : \mathsf{SynLP}_{\mathbb{L}} \to \mathbf{Set}(\mathcal{K})$. On objects, $[\![A]\!]_{\mathbb{P}} := \mathcal{K}_A$ for each $A \in At$. On morphisms, for the CDMU structure we define

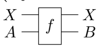
$$\begin{array}{llll}
[\![{-}\!\!\bullet\!\!{<}_A]\!]_{\mathbb{P}} : \mathcal{K}_A \to \mathcal{K}_A \times \mathcal{K}_A & [\![{>}\!\!\bullet\!\!{-}_A]\!]_{\mathbb{P}} : \mathcal{K}_A \times \mathcal{K}_A \to \mathcal{K}_A & [\![{-}\!\!\bullet_A]\!]_{\mathbb{P}} : \mathcal{K}_A \to \mathbf{1} & [\![\bullet\!\!{-}_A]\!]_{\mathbb{P}} : \mathbf{1} \to \mathcal{K}_A \\
\qquad x \mapsto (x, x) & \qquad (x, y) \mapsto x + y & \qquad x \mapsto * & \qquad * \mapsto \mathbf{0}_A
\end{array}$$

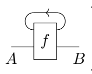
For each generating morphism ${}^{B_1}_{B_m}\!\vdots\boxed{\varphi}{-}_A$ in $\Sigma_{\mathbb{L}}$, $[\![\varphi]\!]_{\mathbb{P}}$ maps $u = (u_1, \ldots, u_m) \in \mathcal{K}_{B_1} \times \cdots \times \mathcal{K}_{B_m}$ to $lab(\psi) \cdot u_1 \cdot \cdots \cdot u_m$, where ψ is the (unique) $\mathbb{P}$-clause satisfying $\tau^{wt}(\psi) = \varphi$. $\mathbf{Set}(\mathcal{K})$ being a CDMU category (Lemma 29, Appendix B) guarantees that $[\![-]\!]_{\mathbb{P}}$ is well-defined.

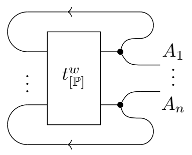
▶ **Proposition 25.** *There is a 1-1 correspondence between weighted logic programs based on* $\mathbb{L}$ *and generator-preserving CDMU functors* $\mathsf{SynLP}_{\mathbb{L}} \to \mathbf{Set}(\mathcal{K})$.

Analogously to what we did with LP and PLP, we conclude the section by describing how the semantics of WLP (see Sec. 2) can be expressed with string diagrams of the syntax category.

Let us fix a WLP $\mathbb{P}$, whose atom set is $At = \{A_1, \dots, A_n\}$. First, the string diagram $t^w_{[\mathbb{P}]}$ expressing the weighted immediate consequence operator $\mathbf{T}^w_{\mathbb{P}}$ (*cf.* Section 2) is defined as in (4), where $\varphi_1, \dots, \varphi_N$ are the generating morphisms in $\Sigma_{[\mathbb{P}]}$. As expected, $\left[\!\left[t^w_{[\mathbb{P}]} \right]\!\right]_{\mathbb{P}} = \mathbf{T}^w_{\mathbb{P}}$.

As a second step, we want to express the semantics of WLP, as defined in Section 2. As this is a least fixed point semantics, we use the extended syntax $\mathsf{SynLP}^{Tr}_{[\mathbb{P}]} = \mathsf{freeTrCDMU}(At, \Sigma_{[\mathbb{P}]})$ from Section 3.3, and interpret it in the category $\mathbf{Rel}(\mathcal{K})$, whose objects are that of $\mathbf{Set}(\mathcal{K})$ and morphisms $A \to B$ are relations $R \subseteq A \times B$. The program $\mathbb{P}$ uniquely determines a traced CDMU functor $[\![-]\!]^{Tr}_{\mathbb{P}} : \mathsf{SynLP}^{Tr}_{[\mathbb{P}]} \to \mathbf{Rel}(\mathcal{K})$ inductively defined on the generating morphisms, in a similar manner as that for classical logic program (*cf.* Section 3.3). In particular, the inductive step includes that, for each string diagram  in $\mathsf{SynLP}^{Tr}_{[\mathbb{P}]}$,

$$\left[\!\left[\text{} \right]\!\right]^{Tr}_{\mathbb{P}} = \{(a,b) \in [\![A]\!]^{Tr}_{\mathbb{P}} \times [\![B]\!]^{Tr}_{\mathbb{P}} \mid \exists x \in [\![X]\!]^{Tr}_{\mathbb{P}} : ((x,a),(x,b)) \in [\![f]\!]^{Tr}_{\mathbb{P}}\}.$$



Since atom weights $(weight_{\mathbb{P}}(A_1), \dots, weight_{\mathbb{P}}(A_n))$ are the least fixed point of $\mathbf{T}^w_{\mathbb{P}}$ (*cf.* Section 2), we can express them as $t^w_{[\mathbb{P}]}$ with 'feedback' wires, resulting in the string diagram $W_{[\mathbb{P}]}$ defined as on the right. Formally, this means that $(\min [\![W_{[\mathbb{P}]}]\!]^{Tr}_{\mathbb{P}})(i) = weight_{\mathbb{P}}(A_i)$, for $i = 1, \dots, n$.

6 Conclusions

In this paper we established a functorial perspective on logic programming, encompassing classical, probabilistic, and weighted programs. This allowed us to propose an original viewpoint on some well-known semantic constructs for these formalisms, and on the correspondence between probabilistic programs and Bayesian networks.

This work paves the way for several future developments:

- The functorial view on the equivalence between probabilistic programs and Bayesian networks provides a basis for a comparative analysis of various inference tasks in logic programming with inference in Bayesian reasoning, for which we may rely on several recent categorical approaches [11, 23, 24, 22, 21]. In particular, the maximum a posteriori (MAP) task [5], most probable explanation (MPE) task [5], and inductive logic programming (ILP) [32] seem mostly promising.
- We would like to extend the equivalence between probabilistic programs and Bayesian networks to richer classes. On the side of logic programming, this includes CP-logic [39] and logic programs with annotated disjunctions (LPAD) [40]. On the side of Bayesian networks, we may study causal diagrams for structural equation models (SEM) [34].
- For simplicity, this paper only deals with the ground case: all the logic programs we consider are propositional, without variables. We leave a functorial semantics of arbitrary logic programs (with variables) as future work, potentially using more sophisticated formalisms of string diagrams, such as nominal diagrams [4].
- Classical, probabilistic and weighted logic programming already attracted a categorical modelling, in terms of coalgebras [27, 26, 7, 6, 19]. A natural research direction is exploring possible synergies between these works and ours. In particular, a string diagrammatic analysis of coinductive logic programming [28, 20] seems particularly intriguing.

References

1 Samson Abramsky. Abstract scalars, loops, and free traced and strongly compact closed categories. volume 3629, 10 2009. doi:10.1007/11548133_1.

2 Joyal Andre, Ross Street, and Dominic Verity. Traced monoidal categories. *Mathematical Proceedings of the Cambridge Philosophical Society*, 119:447 – 468, 04 1996. doi:10.1017/S0305004100074338.

3 Krzysztof R Apt, Howard A Blair, and Adrian Walker. Towards a theory of declarative knowledge. In *Foundations of deductive databases and logic programming*, pages 89–148. Elsevier, 1988.

4 Samuel Balco and Alexander Kurz. Nominal string diagrams. In Markus Roggenbach and Ana Sokolova, editors, *8th Conference on Algebra and Coalgebra in Computer Science, CALCO 2019, June 3-6, 2019, London, United Kingdom*, volume 139 of *LIPIcs*, pages 18:1–18:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. URL: https://doi.org/10.4230/LIPIcs.CALCO.2019.18, doi:10.4230/LIPIcs.CALCO.2019.18.

5 Elena Bellodi, Marco Alberti, Fabrizio Riguzzi, and Riccardo Zese. Map inference for probabilistic logic programming. *Theory and Practice of Logic Programming*, 20(5):641–655, 2020.

6 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. URL: https://doi.org/10.1007/978-3-642-40206-7_8, doi:10.1007/978-3-642-40206-7_8.

7 Filippo Bonchi and Fabio Zanasi. Bialgebraic semantics for logic programming. *Logical Methods in Computer Science*, 11(1), 2015. URL: https://doi.org/10.2168/LMCS-11(1:14)2015, doi:10.2168/LMCS-11(1:14)2015.

8 Ivan Bratko. *Prolog programming for artificial intelligence*. Pearson education, 2001.

9 Stefano Ceri, Georg Gottlob, and Letizia Tanca. *Logic programming and databases*. Springer Science & Business Media, 2012.

10 Shay Cohen, Robert Simmons, and Noah Smith. Products of weighted logic programs. *Computing Research Repository - CORR*, 11, 06 2010. doi:10.1017/S1471068410000529.

11 Jared Culbertson and Kirk Sturtz. A categorical foundation for bayesian probability. *Applied Categorical Structures*, 22(4):647–662, Aug 2013. URL: http://dx.doi.org/10.1007/s10485-013-9324-9, doi:10.1007/s10485-013-9324-9.

12 Eduardo Menezes de Morais and Marcelo Finger. Probabilistic answer set programming. In *2013 Brazilian Conference on Intelligent Systems*, pages 150–156. IEEE, 2013.

13 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. URL: http://dl.acm.org/citation.cfm?id=1625275.1625673.

14 Didier Dubois, Lluas Godo, and Henri Prade. Weighted logics for artificial intelligence : an introductory discussion. *International Journal of Approximate Reasoning*, 55(9):1819 – 1829, 2014. Weighted Logics for Artificial Intelligence. doi:https://doi.org/10.1016/j.ijar.2014.08.002.

15 Jason Eisner and John Blatz. Program transformations for optimization of parsing algorithms and other weighted logic programs. In Shuly Wintner, editor, *Proceedings of FG 2006: The 11th Conference on Formal Grammar*, pages 45–85. CSLI Publications, 2007. URL: http://cs.jhu.edu/~jason/papers/#eisner-blatz-2007.

16 Brendan Fong. Causal theories: A categorical perspective on bayesian networks. 2013. arXiv:1301.6201.

17 Michael Gelfond. On stratified autoepistemic theories. In *AAAI*, volume 87, pages 207–211, 1987.

18 Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, volume 88, pages 1070–1080, 1988.

19 Tao Gu and Fabio Zanasi. Coalgebraic Semantics for Probabilistic Logic Programming. *Logical Methods in Computer Science*, Volume 17, Issue 2, April 2021. URL: https://lmcs.episciences.org/7365.

20 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.

21 Bart Jacobs. A channel-based exact inference algorithm for bayesian networks. *CoRR*, abs/1804.08032, 2018. URL: http://arxiv.org/abs/1804.08032, arXiv:1804.08032.

22 Bart Jacobs, Aleks Kissinger, and Fabio Zanasi. Causal inference by string diagram surgery. In Mikołaj Bojańczyk and Alex Simpson, editors, *Foundations of Software Science and Computation Structures*, pages 313–329, Cham, 2019. Springer International Publishing.

23 Bart Jacobs and Fabio Zanasi. A predicate/state transformer semantics for bayesian learning. *Electronic Notes in Theoretical Computer Science*, 325:185–200, 2016. The Thirty-second Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXII). doi:https://doi.org/10.1016/j.entcs.2016.09.038.

24 Bart Jacobs and Fabio Zanasi. A Formal Semantics of Influence in Bayesian Reasoning. In Kim G. Larsen, Hans L. Bodlaender, and Jean-Francois Raskin, editors, *42nd International Symposium on Mathematical Foundations of Computer Science (MFCS 2017)*, volume 83 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:14, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. URL: http://drops.dagstuhl.de/opus/volltexte/2017/8089, doi:10.4230/LIPIcs.MFCS.2017.21.

25 Bart Jacobs and Fabio Zanasi. The logical essentials of bayesian reasoning. *arXiv preprint arXiv:1804.01193*, 2018.

26 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. URL: https://doi.org/10.1007/978-3-642-22944-2_19, doi:10.1007/978-3-642-22944-2_19.

27 Ekaterina Komendantskaya and John Power. Logic programming: Laxness and saturation. *J. Log. Algebraic Methods Program.*, 101:1–21, 2018. URL: https://doi.org/10.1016/j.jlamp.2018.07.004, doi:10.1016/j.jlamp.2018.07.004.

28 Ekaterina Komendantskaya, John Power, and Martin Schmidt. Coalgebraic logic programming: from semantics to implementation. *J. Log. Comput.*, 26(2):745–783, 2016. URL: https://doi.org/10.1093/logcom/exu026, doi:10.1093/logcom/exu026.

29 F William Lawvere. Functorial semantics of algebraic theories. *Proceedings of the National Academy of Sciences of the United States of America*, 50(5):869, 1963.

30 Vladimir Lifschitz. On the declarative semantics of logic programs with negation. In *Foundations of deductive databases and logic programming*, pages 177–192. Elsevier, 1988.

31 Wannes Meert, Jan Struyf, and Hendrik Blockeel. Learning ground cp-logic theories by leveraging bayesian network learning techniques. *Fundam. Inform.*, 89:131–160, 01 2008.

32 Stephen Muggleton and Luc de Raedt. Inductive logic programming: Theory and methods. *The Journal of Logic Programming*, 19-20:629 – 679, 1994. Special Issue: Ten Years of Logic Programming. URL: http://www.sciencedirect.com/science/article/pii/0743106694900353, doi:https://doi.org/10.1016/0743-1066(94)90035-3.

33 Ulf Nilsson and Jan Małuszyński. *Logic, programming and Prolog.* Wiley Chichester, 1990.

34 Judea Pearl. *Causality.* Cambridge university press, 2009.

35 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.

36 Taisuke Sato and Yoshitaka Kameya. New advances in logic-based probabilistic modeling by prism. In *Probabilistic inductive logic programming*, pages 118–155. Springer, 2008.

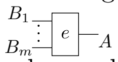
37 P. Selinger. *A Survey of Graphical Languages for Monoidal Categories*, pages 289–355. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. URL: https://doi.org/10.1007/978-3-642-12821-9_4.

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

39 Joost Vennekens, Marc Denecker, and Maurice Bruynooghe. Representing causal information about a probabilistic process. In Michael Fisher, Wiebe van der Hoek, Boris Konev, and Alexei Lisitsa, editors, *Logics in Artificial Intelligence*, pages 452–464, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

40 Joost Vennekens, Sofie Verbaeten, and Maurice Bruynooghe. Logic programs with annotated disjunctions. In *Logic Programming*, pages 409–415, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. doi:10.1007/978-3-540-27775-0_30.

A Functorial semantics of Bayesian networks

We briefly recall the functorial semantics for Bayesian networks from [22]. The semantics category is the (CD) category **Stoch** of finite sets and stochastic processes. Given a Bayesian network $\mathbb{B} = (G, \Pr)$, the syntax category SynBN_G the freely generated CD category $\mathsf{freeCD}(V_G, \Sigma_G)$, where $\Sigma_G :=$ $\left\{ \right.$  $\left| \; A \in V_G \text{ and } pa(A) = \{B_1, \dots, B_m\} \right\}$.
The functorial semantics of Bayesian networks can be precisely stated as:

▶ **Proposition 26** ([22], Proposition 3.1). *There is 1-1 correspondence between Bayesian networks based on a DAG G and CD functors $\mathsf{SynBN}_G \to$ **Stoch**.*

Proposition 19, the functorial semantics of boolean-valued Bayesian networks, follows immediately from Proposition 26 by restriction to boolean-valued Bayesian networks.

B Omitted proofs

▶ **Lemma 27.** $\mathbf{Set}(\mathbf{2})$ *is a CDMU category.*

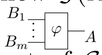
Proof. The category $\langle \mathbf{Set}(\mathbf{2}), \times, \mathbf{1} \rangle$ is a SMC category, where $\times$ is the cartesian product in **Set**, and $\mathbf{1}$ is the singleton set $\{*\}$ (which is the 0-ary product of copies of $\mathbf{2}$). We define the CDMU structure on the two-element boolean algebra $\mathbf{2} = \{0, 1\}$, and that for a copy $\mathbf{2}_A$ follows by replacing 0 and 1 with $\neg A$ and A, respectively:

$$\begin{array}{llll} -\!\!\bullet\!\!<_{\mathbf{2}} : \mathbf{2} \to \mathbf{2} \times \mathbf{2} & >\!\!\bullet\!\!-_{\mathbf{2}} : \mathbf{2} \times \mathbf{2} \to \mathbf{2} & -\!\!\bullet_{\mathbf{2}} : \mathbf{2} \to \mathbf{1} & \bullet\!\!-_{\mathbf{2}} : \mathbf{1} \to \mathbf{2} \\ \quad x \mapsto (x, x) & \quad (x, y) \mapsto x \vee y & \quad x \mapsto * & \quad * \mapsto 0 \end{array}$$

We only verify the equations for the monoid structure. Given arbitrary $x, y, z \in \mathbf{2}$,
- $((id \otimes \bullet\!\!-); >\!\!\bullet\!\!-)(x) = >\!\!\bullet\!\!-(x, 0) = x \vee 0 = x$. Similarly $(\bullet\!\!- \otimes id); >\!\!\bullet\!\!- = id$.
- $((>\!\!\bullet\!\!- \otimes id); >\!\!\bullet\!\!-)(x, y, z) = >\!\!\bullet\!\!-(x \vee y, z) = x \vee y \vee z = ((id \otimes >\!\!\bullet\!\!-); >\!\!\bullet\!\!-)(x, y, z)$.
- $(\times; >\!\!\bullet\!\!-)(x, y) = >\!\!\bullet\!\!-(y, x) = x \vee y = >\!\!\bullet\!\!-(x, y)$.

The CDMU structure on arbitrary objects of the form $\mathbf{2}_{A_1} \times \cdots \times \mathbf{2}_{A_k}$ are defined pointwise, and it follows immediately that they satisfy the CDMU equations. ◀

Proof of Proposition 5. The construction of a CDMU functor $[\![-]\!]_{\mathbb{P}}$ from a program $\mathbb{P}$ is already discussion in Subsection 4.1.

For the other direction, given a generator-preserving CDMU functor $\mathcal{G}\colon \mathsf{SynLP}_{\mathbb{L}} \to \mathbf{Set}(\mathbf{2})$, we know $\mathcal{G}(A)$ is some copy of $\mathbf{2}$, say $\mathbf{2}_A$, for each $A \in At$. For each generating morphism  in $\Sigma_{\mathbb{L}}$, $\mathcal{G}(\varphi)$ is a function $f\colon \mathbf{2}_{B_1} \times \cdots \times \mathbf{2}_{B_m} \to \mathbf{2}_A$. Suppose that the support of $\mathcal{G}(\varphi)$ — the inverse image of $A \in \mathbf{2}A$ under f — is $\{v_1, \dots, v_d\}$. Then we construct d-many clauses $\varphi_1, \dots, \varphi_d$ such that $\tau^{gen}(\varphi_j) = A \leftarrow B_1, \dots, B_m.$, one for each v_i. $v_i(B_{p_1}) = B_{p_1}, \cdots = v_i(B_{p_k}) = B_{p_k}$, $v_i(B_{q_1}) = \neg B_{q_1}, \dots, v_i(B_{q_\ell}) = \neg B_{q_\ell}$ (with $k + \ell = m$ and $\{p_1, \dots, p_k, q_1, \dots, q_\ell\} = \{1, \dots, m\}$), then let clause φ_i be $A \leftarrow B_{p_1}, \dots, B_{p_k}, \neg B_{q_1}, \dots, \neg B_{q_\ell}.$. The program $\langle \mathcal{G} \rangle$ is then the collection of all the clauses constructed from each generating morphism in $\Sigma_{\mathbb{L}}$.

We show that the aforementioned two constructions are inverse to each other.
- We start from a logic program $\mathbb{P}$ based on $\mathsf{SynLP}_{\mathbb{L}}$. For an arbitrary $\varphi \equiv A \leftarrow B_1, \dots, B_m.$ in $\mathbb{L}$, suppose $\{\varphi_1, \dots, \varphi_d\}$ are all the clauses φ_j such that $\tau^{gen}(\varphi_j) = \varphi$, then $[\![\varphi]\!]_{\mathbb{P}}$ is defined as a function $\mathbf{2}_{B_1} \times \cdots \times \mathbf{2}_{B_m} \to \mathbf{2}_A$ whose support (namely $\mathcal{G}(\varphi)^{-1}(A)$) has size d. Then from $[\![\varphi]\!]_{\mathbb{P}}$ we retrieve d clauses, which are exactly $\varphi_1, \dots, \varphi_d$: if $v \in \mathbf{2}_{B_1} \times \cdots \times \mathbf{2}_{B_m}$ satisfies $[\![\varphi]\!]_{\mathbb{P}}(v) = A$, then there exists some φ_j such that for all $i = 1, \dots, m$, $v(B_i) = B_i$ if and only if B_i appears positively (namely as B) in $\mathsf{body}(\varphi_j)$, and the clause induced by v is exactly φ_j itself.

- We start from a functor $\mathcal{G}\colon \mathsf{SynLP}_{\mathbb{L}} \to \mathbf{Set}(\mathbf{2})$. Fix some $\mathbb{L}$-clause φ, and suppose $\mathcal{G}(\varphi)^{-1}(A) = \{v_1, \dots, v_d\}$. Then $\mathcal{G}$ determines d-many clauses $\varphi_1, \dots, \varphi_d$ whose image under τ^{gen} is φ. Then the action of the functor $[\![-]\!]_{\langle\mathcal{G}\rangle} : \mathsf{SynLP}_{\mathbb{L}} \to \mathbf{Set}(\mathbf{2})$ on $\varphi \in \mathbb{L}$ is totally determined by $\{\varphi_1, \dots, \varphi_d\}$: $[\![\varphi]\!]_{\langle\mathcal{G}\rangle}(v) = A$ if and only if v is compatible with $\mathsf{body}(\varphi_j)$ for some $\varphi_j \in \{\varphi_1, \dots, \varphi_d\}$; but φ_j is exactly represented by v_j, so $[\![\varphi]\!]_{\langle\mathcal{G}\rangle}(v) = A$ if and only if $v = v_j$, for some $v_j \in \{v_1, \dots, v_d\}$. This means that $[\![\varphi]\!]_{\langle\mathcal{G}\rangle} = \mathcal{G}(\varphi)$, for arbitrary $\mathbb{L}$-clause φ.

Therefore there is a bijection between logic programs based on $\mathbb{L}$ and generator-preserving CDMU functors $\mathsf{SynLP}_{\mathbb{L}} \to \mathbf{Set}(\mathbf{2})$. ◀

Proof of Proposition 7. We are given a logic program $\mathbb{P}$ based on $\mathbb{L}$ with atom set $At = \{A_1, \dots, A_n\}$. For arbitrary $u \in \mathbf{2}_{A_1} \times \dots \times \mathbf{2}_{A_n}$ and $A \in At$, we know $\mathbf{T}_{\mathbb{P}}(u)(A) = A$ if and only if there exists $\psi \in \mathbb{P}$ such that $\mathsf{head}(\psi) = A$ and $u \vDash \mathsf{body}(\psi)$. Let $\varphi = \tau^{gen}(\psi)$, then $u \vDash \mathsf{body}(\psi)$ if and only if $[\![\varphi]\!]_{\mathbb{P}} : u|_{\varphi} \mapsto A$, where $u|_{\varphi}$ is the projection of u (as a tuple of values) to the set of atoms in $\mathsf{body}(\varphi)$. So $\mathbf{T}_{\mathbb{P}}(u)(A) = A$ if and only if there exists $\varphi \in [\mathbb{P}]$ such that $\mathsf{head}(\varphi) = A$ and $[\![\varphi]\!]_{\mathbb{P}} : u|_{\varphi} \mapsto A$. By the interpretation of $-\!\!\blacktriangleleft$ and $\blacktriangleright\!\!-$ under $[\![-]\!]_{\mathbb{P}}$, this again is equivalent to that $[\![t_{[\mathbb{P}]}]\!]_{\mathbb{P}}(u)(A) = A$. ◀

Proof of Proposition 10 . It suffices to show that $[\![h_{[\mathbb{P}]}]\!]_{\mathbb{P}}^{Tr}$ is exactly the set of all fixed points of $\mathbf{T}_{\mathbb{P}}$. On one hand, suppose u is a fixed point of $\mathbf{T}_{\mathbb{P}}$, then by Proposition 7 $[\![t_{[\mathbb{P}]}]\!]_{\mathbb{P}}^{Tr}(u) = \mathbf{T}_{\mathbb{P}}(u) = u$. It follows from the definition of $[\![-]\!]_{\mathbb{P}}^{Tr}$ on traced morphisms that u is an element in $[\![h_{[\mathbb{P}]}]\!]_{\mathbb{P}}^{Tr}$. On the other hand, let v be a state in $[\![h_{[\mathbb{P}]}]\!]_{\mathbb{P}}^{Tr}$, then by definition of $[\![-]\!]_{\mathbb{P}}^{Tr}$ on traced morphisms, $[\![t_{[\mathbb{P}]}]\!]_{\mathbb{P}}^{Tr}(v) = v$, so $\mathbf{T}_{\mathbb{P}}(v) = v$. ◀

Proof of Proposition 13. The direction from a PLP $\mathbb{P}$ to a generator-preserving CDMU functor $[\![-]\!]_{\mathbb{P}}$ is already discussed in Subsection 4.1.

For the other direction, given a generator-preserving CDMU functor $\mathcal{G}\colon \mathsf{SynLP}_{\mathbb{L}} \to \mathbf{Stoch}(\mathbf{2})$, it maps each $A \in At$ to a copy of $\mathbf{2}$, say $\mathbf{2}_A$. We define a PLP program $\langle\mathcal{G}\rangle$. For an arbitrary ${}^{B_1}_{B_m}\!\vdots\boxed{\varphi}\!-_A$ in $\Sigma_{\mathbb{L}}$, $\mathcal{G}(\varphi)$ is a function $\mathbf{2}_{B_1} \times \dots \times \mathbf{2}_{B_m} \to \mathcal{D}(\mathbf{2}_A)$. Let $\{v_1, \dots, v_d\}$ be the set of all $v \in \mathbf{2}_{B_1} \times \dots \times \mathbf{2}_{B_m}$ such that $\mathcal{G}(\varphi)(v) \neq 1|\neg A\rangle$. Then we construct one PLP clause for every such v: suppose $v(B_{i_1}) = B_{i_1}, \dots, v(B_{i_k}) = B_{i_k}$, $v(B_{j_1}) = \neg B_{j_1}, \dots, v(B_{j_\ell}) = \neg B_{j_\ell}$ be an enumeration of all the components of the m-tuple v, and $\mathcal{G}(\varphi)(v) = p|A\rangle + (1-p)|\neg A\rangle$, then we define a clause $\varphi_v \equiv p :: A \leftarrow B_{i_1}, \dots, B_{i_k}, \neg B_{j_1}, \dots, \neg B_{j_\ell}$. The PLP program $\langle\mathcal{G}\rangle$ consists of all the clauses $\{\varphi_{v_1}, \dots, \varphi_{v_d}\}$ defined as above for each φ in $\Sigma_{\mathbb{L}}$.

The above two procedures $[\![-]\!]_{(\cdot)}$ and $\langle\cdot\rangle$ are inverse to each other. Starting from a $\mathbb{P}$-clause $\psi \equiv p :: A \to B_{i_1}, \dots, B_{i_k}, \neg B_{j_1}, \dots, \neg B_{j_\ell}$. with $\tau^{pr}(\psi) = \varphi$, ψ determines the behaviour of the functor $[\![\varphi]\!]_{\mathbb{P}}$ at the unique input v with $v \vDash \mathsf{body}(\psi)$ as $[\![\varphi]\!]_{\mathbb{P}}(v) = p|A\rangle + (1-p)|\neg A\rangle$. This defines a unique clause in $\langle[\![-]\!]_{\mathbb{P}}\rangle$, which is exactly ψ. Starting from a functor $\mathcal{G}$, it generates 2^m clauses in $\langle\mathcal{G}\rangle$ from each ${}^{B_1}_{B_m}\!\vdots\boxed{\varphi}\!-_A$ in $\Sigma_{\mathbb{L}}$, all of whose τ^{pr}-images are ${}^{B_1}_{B_m}\!\vdots\boxed{\varphi}\!-_A$. This behaviour of $[\![-]\!]_{\langle\mathcal{G}\rangle}$ on ${}^{B_1}_{B_m}\!\vdots\boxed{\varphi}\!-_A$ is determined by these 2^m clauses, which is exactly $\mathcal{G}({}^{B_1}_{B_m}\!\vdots\boxed{\varphi}\!-_A)$. ◀

Proof of Proposition 17. It suffices to show $[\![f_{At}]\!]_{\mathbb{P}} = \delta_{\mathbb{P}}(At)$, and that for subsets of At follows by observing that using string diagrams, marginal distributions can be obtained from the joint distributions by applying discarders $-\!\bullet$ [25]. We prove $[\![f_{At}]\!]_{\mathbb{P}} = \delta_{\mathbb{P}}(At)$ by induction on the size of At. If $|At| = 1$, then $\mathbb{P}$ can only contain one clause of the form $\psi \equiv p :: A \leftarrow .$, so $[\![f_{At}]\!]_{\mathbb{P}} = p|A\rangle + (1-p)|\neg A\rangle$, which corresponds to that $\pi_{\mathbb{P}}(A) = p$, $\pi_{\mathbb{P}}(\neg A) = 1 - p$.

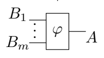
Now suppose $|At| = n + 1$ and the proposition holds for all programs with atom size $\leq n$. Because $\mathbb{P}$ is acyclic, we can select an atom $A \in At$ such that there is no $\mathbb{P}$-clause φ with

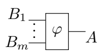
$A \in \mathsf{body}(\varphi)$. Let $\mathbb{P}^A$ be the program consisting of all $\mathbb{P}$-clauses with head A, $\mathbb{P}^-$ be $\mathbb{P} \setminus \mathbb{P}^A$, and At^- be $At \setminus \{A\}$. If $\mathbb{P}^A = \varnothing$, then there is no $\mathbb{P}$-clause with head A, $f_{At} = f_{At^-} \otimes \bullet\!\!-_A$. Thus $[\![f_{At}]\!]_{\mathbb{P}} = \sum_{u \in \mathbf{2}_{At^-}} \delta_{\mathbb{P}^-}(u)|u, 0_A\rangle = \delta_{\mathbb{P}}$

So we assume $\mathbb{P}^A \neq \varnothing$. For each interpretation $\mathcal{I}$ of $\mathbb{P}$, we observe that if $A \in \mathcal{I}$, then a sub-program $\mathbb{L} \subseteq |\mathbb{P}|$ satisfies $M_l(\mathbb{L}) = \mathcal{I}$ if and only if $M_l(\mathbb{L}^-) = \mathcal{I}^-$ and there exists $\varphi \in \mathbb{L}^A$ such that $\mathcal{I} \vDash \mathsf{body}(\varphi)$; if $A \notin \mathcal{I}$, then a sub-program $\mathbb{L} \subseteq |\mathbb{P}|$ satisfies $M_l(\mathbb{L}) = \mathcal{I}$ if and only if $M_l(\mathbb{L}^-) = \mathcal{I}^-$ and $\mathcal{I} \nvDash \mathsf{body}(\varphi)$ for all $\varphi \in \mathbb{L}^A$. We assume that $comp_A$ is of the form $[B_1, \dots, B_k] \to [A]$. We can calculate $\delta_{\mathbb{P}}(At)$ using the induction hypothesis $\delta_{\mathbb{P}^-}(At^-)(\mathcal{I}^-) = [\![f_{At^-}]\!]_{\mathbb{P}}(\mathcal{I}^-)$. For example,

$$
\begin{aligned}
&\delta_{\mathbb{P}}(At)(\mathcal{I}^- \cup \{A\}) \\
={}& \sum \{\mu_{\mathbb{P}}(\mathbb{L}) \mid \mathbb{L} \subseteq |\mathbb{P}|, M_l(\mathbb{L}) = \mathcal{I}^- \cup \{A\}\} \\
={}& \sum \{\mu_{\mathbb{P}^-}(\mathbb{L}^-) \cdot \mu_{\mathbb{P}^A}(\mathbb{L}^A) \mid \mathbb{L}^- \subseteq |\mathbb{P}^-|, \mathbb{L}^A \subseteq |\mathbb{P}^A|, M_l(\mathbb{L}^-) = \mathcal{I}^-, \exists \varphi \in \mathbb{P}^A \text{ s.t. } \mathcal{I}^- \vDash \mathsf{body}(\varphi)\} \\
={}& \sum \{\mu_{\mathbb{P}^-}(\mathbb{L}^-) \cdot [\![comp_A]\!]_{\mathbb{P}}(\mathcal{I}^-|_{B_1,\dots,B_k})(A) \mid \mathbb{L}^- \subseteq |\mathbb{P}^-|, M_l(\mathbb{L}^-) = \mathcal{I}^-\} \\
={}& \delta_{\mathbb{P}^-}(At^-)(\mathcal{I}^-) \cdot [\![comp_A]\!]_{\mathbb{P}}(\mathcal{I}^-|_{B_1,\dots,B_k})(A) \\
\overset{\text{IH}}{=}{}& [\![f_{At^-}]\!]_{\mathbb{P}}(\mathcal{I}^-) \cdot [\![comp_A]\!]_{\mathbb{P}}(\mathcal{I}^-|_{B_1,\dots,B_k})(A) \\
={}& [\![f_{At}]\!]_{\mathbb{P}}(\mathcal{I}^- \cup \{A\})
\end{aligned}
$$

Similarly we can show $\delta_{\mathbb{P}}(At)(\mathcal{I}^-) = [\![f_{At}]\!]_{\mathbb{P}}(\mathcal{I}^-)$. Since every interpretation $\mathcal{I}$ for $\mathbb{P}$ can be divided as an interpretation $\mathcal{I}^-$ on $\mathbb{P}^-$ and a subset of $\{A\}$, it follows that $\delta_{\mathbb{P}} = [\![f_{At}]\!]_{\mathbb{P}}$. ◀

Proof of Proposition 20. For each node A, suppose $pa(A) = \{B_1, \dots, B_m\}$, then there are exactly 2^m clauses ψ_i in $\mathbb{P}$ whose heads are A (possibly some ψ_i has probability label 0), each satisfying $\tau^{npr}(\psi_i) = A \leftarrow B_1, \dots, B_m$.. By (3), $\Sigma_{[\mathbb{P}]} = \Big\{$  $\Big|\ pa(A) = \{B_1, \dots, B_m\}\Big\} = \Sigma_{\mathbb{L}}$.

To show that $[\![-]\!]_{\mathbb{P}} = [\![-]\!]^b_{\mathbb{B}}$, it suffices to show that they coincide on the generating morphisms $\Sigma_{\mathbb{L}}$. Given arbitrary  $\in \Sigma_{\mathbb{B}}$ and state $u \in \mathbf{2}_{B_1} \times \cdots \times \mathbf{2}_{B_m}$, we show that $[\![\varphi]\!]_{\mathbb{P}}(u) = [\![\varphi]\!]^b_{\mathbb{B}}(u)$. There is exactly one $\mathbb{P}$-clause ψ satisfying both $\tau^{pr}(\psi) = \varphi$ and $u \vDash \mathsf{body}(\psi)$. Suppose the corresponding interpretation of u is $\{B_{d_1}, \dots, B_{d_s}\}$, whose complement regarding $\{B_1, \dots, B_m\}$ is $\{B_{e_1}, \dots, B_{e_t}\}$, then

$$[\![\varphi]\!]_{\mathbb{P}}(u) \coloneqq lab(\psi) = \Pr(A \mid B_{d_1}, \dots, B_{d_s}, \neg B_{e_1}, \dots, \neg B_{e_t})$$

This coincides with the definition of $[\![\varphi]\!]_{\mathbb{B}}$, thus of $[\![\varphi]\!]^b_{\mathbb{B}}$, on u. ◀

▶ **Lemma 28.** *$\mathbf{Stoch}(\mathbf{2})$ is a CDMU category.*

Proof. First of all, $\langle \mathbf{Stoch}(\mathbf{2}), \times, \mathbf{1}\rangle$ forms a SMC [22]. We define the CDMU structure on $\mathbf{2}$, and that for each copy $\mathbf{2}_A$ follows by replacing 1 and 0 with A and $\neg A$, respectively.

$$
\begin{array}{llll}
-\!\!\bullet\!\!<_{\mathbf{2}} : \mathbf{2} \to \mathbf{2} \times \mathbf{2} & >\!\!\bullet\!\!-_{\mathbf{2}} : \mathbf{2} \times \mathbf{2} \to \mathbf{2} & -\!\!\bullet_{\mathbf{2}} : \mathbf{2} \to \mathbf{1} & \bullet\!\!-_{\mathbf{2}} : \mathbf{1} \to \mathbf{2} \\
\quad x \mapsto 1|(x,x)\rangle & \quad (x,y) \mapsto 1|x \vee y\rangle & \quad x \mapsto 1|*\rangle & \quad * \mapsto 1|0\rangle
\end{array}
$$

We verify the CDMU equations for the monoid structure, and that for the comonoid structure can be found in [22]. Given arbitrary $x, y, z \in \mathbf{2}$,

- $((id \otimes \bullet\!\!-); >\!\!\bullet\!\!-)(x) = 1|x \vee 0\rangle = 1|x\rangle$. Similarly $(\bullet\!\!- \otimes id); >\!\!\bullet\!\!- = id$.
- $((>\!\!\bullet\!\!- \otimes id); >\!\!\bullet\!\!-)(x, y, z) = 1|x \vee y \vee z\rangle = ((id \otimes >\!\!\bullet\!\!-); >\!\!\bullet\!\!-)(x, y, z)$.
- $(\times; >\!\!\bullet\!\!-)(x, y) = 1|x \vee y\rangle = >\!\!\bullet\!\!-(x, y)$.

The CDMU structure on arbitrary objects of the form $\mathbf{2}_{A_1} \times \cdots \times \mathbf{2}_{A_k}$ is defined pointwise, and it follows immediately that the structure satisfies CDMU equations. ◀

Proof of Proposition 23. The equivalence of the set of generating morphisms follows immediately from their definitions. To show that $[\![-]\!]_{\mathbb{B}} = [\![-]\!]_{\mathbb{P}} \circ \mathcal{F}$, it suffices to show that for arbitrary generating morphism ${}^{B_1}_{B_m}\!\vdots\!\boxed{e}\!-_A$ in Σ_G and state $u = (u_1, \ldots, u_m)$ of $B_1, \ldots, B_m$, $[\![\mathcal{F}(e)]\!]_{\mathbb{P}}(u)$ is equivalent to $\Pr(A = 1 \mid B_1 = x_1, \ldots, B_m = x_m)$.

Suppose $\varphi_1, \ldots, \varphi_k$ are all the $[\mathbb{P}]$-clauses with heads A, so they exhaust all the components of $comp_A$ besides the CDMU structure. For each φ_i of the form $A \leftarrow B_{i_1}, \ldots, B_{i_{m_i}}$, $[\![\varphi_i]\!]_{\mathbb{P}}(u) = p|A\rangle + (1-p)|\neg A\rangle$, where p is the probability label of the $\mathbb{P}$-clause ψ satisfying $[\psi] = \varphi$ and $u \vDash \mathsf{body}(\psi)$, and p is 0 if no such clause exists. Note that if such ψ exists, then it is necessarily unique, so this is well-defined.

Now, suppose we have φ_1, φ_2 with disjoint bodies, and $[\![\varphi_j]\!]_{\mathbb{P}}(u_j) = p_j|A\rangle + (1 - p_j)|\neg A\rangle$, where $u_j = u|_{\mathsf{body}(\varphi_j)}$, $j \in \{1, 2\}$. We compute $[\![(\varphi_i \otimes \varphi_j); \mathord{>\!\!\!-}_A]\!]_{\mathbb{P}}(u')$, where $u' = u|_{\mathsf{body}(\varphi_1) \cup \mathsf{body}(\varphi_2)}$. By the definition of $[\![\mathord{>\!\!\!-}]\!]_{\mathbb{P}}$ and morphisms composition in $\mathbf{Stoch}(\mathbf{2})$, $[\![(\varphi_i \otimes \varphi_j); \mathord{>\!\!\!-}_A]\!]_{\mathbb{P}}(u') = (1-(1-p_1)(1-p_2))|A\rangle + (1-p_1)(1-p_2)|\neg A\rangle$. Then an induction on the number k of components in $comp_A$ shows that, if $[\![\varphi_j]\!]_{\mathbb{P}}(u|_{\mathsf{body}(\varphi_j)}) = p_j|A\rangle + (1-p_j)|\neg A\rangle$, then $[\![comp_A]\!]_{\mathbb{P}}(u) = (1 - \prod_{j=1}^{k}(1-p_j))|A\rangle + \prod_{j=1}^{k}(1-p_j)|\neg A\rangle$. This is exactly the definition of $\Pr(A \mid \{B_i\}_{i=1}^{k} = u)$ in $\mathbb{B}$. ◀

▶ **Lemma 29.** *The category* $\mathbf{Set}(\mathcal{K})$ *is a CDMU category.*

Proof. Category $\mathbf{Set}(\mathcal{K})$ with cartesian product $\times$ and the singleton set $\mathbf{1}$ forms a SMC. Again $\mathbf{1}$ is the 0-ary product of copies of $\mathcal{K}$, thus already included in the definition of $\mathbf{Set}(\mathcal{K})$-objects. We define the CDMU on $\mathcal{K}$, and that on each copy $\mathcal{K}_A$ follows immediately.

$$\begin{array}{rrcl} \mathord{-\!\!\!<}_{\mathcal{K}}: & \mathcal{K} & \to & \mathcal{K} \times \mathcal{K} \\ & x & \mapsto & (x, x) \end{array} \qquad \begin{array}{rrcl} \mathord{>\!\!\!-}_{\mathcal{K}}: & \mathcal{K} \times \mathcal{K} & \to & \mathcal{K} \\ & (x, y) & \mapsto & x \vee y \end{array} \qquad \begin{array}{rrcl} \mathord{-\!\bullet}_{\mathcal{K}}: & \mathcal{K} & \to & \mathbf{1} \\ & x & \mapsto & 1|*\rangle \end{array} \qquad \begin{array}{rrcl} \mathord{\bullet\!-}_{\mathcal{K}}: & \mathbf{1} & \to & \mathcal{K} \\ & * & \mapsto & \mathbf{0} \end{array}$$

We verify the CDMU equations for the monoid structure, and that for the comonoid structure can be found in [22]. Given arbitrary $x, y, z \in \mathcal{K}$,

- $((id \otimes \mathord{\bullet\!-}); \mathord{>\!\!\!-})(x) = \mathord{>\!\!\!-}(x, \mathbf{0}) = x + \mathbf{0} = x$. Similarly $(\mathord{\bullet\!-} \otimes id); \mathord{>\!\!\!-} = id$.
- $((\mathord{>\!\!\!-} \otimes id); \mathord{>\!\!\!-})(x, y, z) = \mathord{>\!\!\!-}(x + y, z) = (x + y) + z = ((id \otimes \mathord{>\!\!\!-}); \mathord{>\!\!\!-})(x, y, z)$.
- $(\mathsf{X}; \mathord{>\!\!\!-})(x, y) = \mathord{>\!\!\!-}(y, x) = x + y = y + x = \mathord{>\!\!\!-}(x, y)$.
- $(\mathord{-\!\!\!<}; (id \otimes \mathord{-\!\bullet}))(x) = (id \otimes \mathord{-\!\bullet})(x, x) = x$. Similarly $(\mathord{-\!\!\!<}; (\mathord{-\!\bullet} \otimes id)) = id$.
- $(\mathord{-\!\!\!<}; (\mathord{-\!\!\!<} \otimes id))(x) = (x, x, x) = \mathord{-\!\!\!<}; (id \otimes \mathord{-\!\!\!<})(x)$.
- $(\mathord{-\!\!\!<}; \mathsf{X})(x) = \mathsf{X}(x, x) = (x, x) = \mathord{-\!\!\!<}(x)$.

The CDMU structure and verification of CDMU equations for general objects of the form $\mathcal{K}_{A_1} \times \mathcal{K}_{A_k}$ follows immediately from a pointwise definition. ◀